REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 1995	

4. TITLE AND SUBTITLE
New Methods in Image Compression Using Multi-Level Transforms and Adaptive Statistical Encoding

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas W. Pike

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Students Attending:
University of Nevada, Las Vegas

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA
95-025

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPARTNEMT OF THE AIR FORCE
AFIT/CI
2950 P STREET, BDLG 125
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW AFR 190-1
Distribution Unlimited
BRIAN D. GAUTHIER, MSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	15. NUMBER OF PAGES 162
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

New Methods in Image Compression Using Multi-Level Transforms and Adaptive Statistical Encoding

by

Thomas W. Pike

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May 1995

The Thesis of Thomas W. Pike for the degree of Master of Science in Computer Science is approved.

Evangelos A. Yfantis April 21, 1995

Chairperson, Evangelos A. Yfantis, Ph. D.

Kia Makki April 24, 1995

Examining Committee, Kia Makki, Ph. D.

John T. Minor

Examining Committee, John T. Minor, Ph. D.

Graduate Faculty Representative, James C. Selser, Ph. D.

Interim Graduate Dean, Cheryl L. Bowles, Ed. D.

University of Nevada, Las Vegas
May 1995

Abstract

The need to meet the demand for high quality digital images, with comparatively modest storage requirements, is driving the development of new image compression techniques. This demand has spurred new techniques based on time to frequency spatial transformation methods. At the core of these methods are a family of transformations built on basis sets called "wavelets." The wavelet transform permits an image to be represented in a substantially reduced space by transferring the energy of the image to a smaller set of coefficients. Although these techniques are lossy as the compression ratio rises, very adequate reconstructions can be made from surprisingly small sets of coefficients. This work explores the transformation process, storage of the representation and the application of these techniques to 24-bit color images. A working color image compression model is illustrated.

Table of Contents

List of Figures

List of Tables

Aknowledgements

I would like to express my debt of gratitude to my advisor and mentor, Dr. Evangelos A. Yfantis, who spent many hours contributing to my understanding of the notion of wavelet theory, his patience and clarity of thought are sources of inspiration for me. I would equally like to thank Blaine Hagstrom, my compatriot graduate student, who shared the many hours of learning with me and served as a sounding board for my ideas. I also would like to thank my wife, Kristine, for without her support and understanding it would have been difficult to devote the time required to make this a reality. Finally, I would like to thank the Air Force Institute of Technology for financially supporting my graduate education.

Chapter 1

Introduction

It is apparent that we are on the verge of another revolution. Not unlike the personal computer industry of only five or ten years ago influencing the way that work is accomplished in the office environment today, we are now facing a leap in the way information is presented to the consumer. This has broad implications across the board. From encyclopedias, magazines and other published media, to interfaces on the Internet, to leisure activities such as games and video taped movies, to the art of photography, are all being influenced by the ability to digitize, store, and retrieve images. Images are the most powerful way to communicate ideas and information to another person, not only by crossing cultural and linguistic barriers, but by providing a common frame of reference for the communication of ideas.

During the past ten years we have increased our ability, at the consumer level, to store and access information by 1,000 folds. At the same time, the demand on our capability to store and access this information has exceeded the 1,000 fold gains. New demands will

probably double or triple the throughput requirements for large amounts of image data in real time applications over the next five years.

A simple example of the kind of requirement needed in the near future is digital video. A broadcast quality video picture might have 512 X 512 pixels and 24 bit (RGB) color resolution. This gives us a digital image of 786,432 bytes with 16.8 million colors. Television pictures are synced at 30 frames per second to give us continuity of motion. So to digitize one second of the video frame signals requires more than 23.5 megabytes of transmission and storage capability. One hour's worth of digitized video would require 84.9 gigabytes of storage and transmission capability. Most personal computers today have 500 megabytes of storage giving a capability of about 22 seconds of raw video storage. The CD-ROM is capable of storing about 650 megabytes of data, but to store an hour of video images would require a compression ratio of over 130:1.

Using this example as a starting point, the digital representation of an image requires a significant number of bytes. The goal of image compression is to reduce this number as much as possible while retaining the ability to restore a faithful reconstruction of the original image. There is a tradeoff involved in the exactness of the reconstruction and level of compression desired. No matter the technique involved in compression there at some point (generally around 2:1 compression ratio) begins a degradation of the restored image. Then of course, a compromise between the acceptable loss of fidelity and achievable compression must be reached.

Every image capturing system, be it a film camera, or a video camera, produces an image by sampling in space, quantizing in brightness, and transforming into a color representation the analog scenes. A photograph does this by modifying the emulsion of the film (actually, the discrete silver halide crystals) and a video camera by digitizing into pixels the sampled values. The sampling step size is usually chosen to be small enough to take advantage of the ability of the human visual system to integrate the data into a continuous image.

Problem Definition

We are therefore faced with a problem on how to meet the demand of efficient storage. Today's JPEG standard does not provide optimal performance as we will see. JPEG [32] stands for the Joint Photographic Experts Group. It is referred as a "joint" group because this committee is sanctioned by the CCITT and the ISO, two prominent international standard groups. JPEG refers both to the committee and their compression standard that defines a method for compressing photographic images. Images compressed with the JPEG algorithm undergo a "lossy" compression. The amount of compression can be varied, with a resultant loss or gain in resolution. JPEG compression can achieve impressive compression ratios compared to the statistical models, reducing the storage required by images to less than 10% of the size of the original with only very slight loss of resolution. By sacrificing more resolution compression can reach to 95% or more using JPEG.

The problem therefore is to develop a compression scheme competitive with JPEG that will restore, with acceptable loss, an image while maximizing its storage. To this end we want to develop a technique that will produce compression ratios of 20:1 or better for color image storage.

Constraints and Assumptions

Let us examine the properties of images and see how this relates to image compression: First, data in an image is not random. Adjacent samples have similar values usually. This correlation or "redundancy" relates to the statistical properties of an image and is a function of resolution, bit-depth, image noise and image detail. If we exploit this redundancy then there is no doubt that we can reduce the number of bits required for the original image. Secondly, there is a component of "irrelevancy" which relates to the observer's comprehension of an image and depends on image noise, detail, and viewing conditions.

We want a method that is flexible enough to manage images of varying sizes and complexity. We will assume that these images convey meaningful information from the average viewer's perspective and are therefore neither white noise nor periodic in nature, but somewhere in between.

First, let us consider the basic representation of data. We will assume a message to be a data set representing either text or an image. Although we are concentrating on image

representation, most of the techniques can also be applied to text. The alphabet of the message is the set of all of the possible symbols that may appear in a message or image. For example, when compressing ASCII text files, the alphabet would probably consist of 127 characters, 0x00 through 0x7f. An image in 8-bit greyscale format would have an alphabet of 256 symbols, 0x00 through 0xff, while a 24-bit RGB color image has an alphabet of nearly 16.8 million symbols.

Compression schemes normally maintain a *model*, which is a set of accumulated statistics describing the state of the encoder. For example, in a simple compression program, the model may be a count of the frequency of every symbol in an input file

What is the minimum number of bits that can be used to represent a message? To put this in context let us examine the notion of entropy. Entropy is the measure of the amount of order in a message, a small value means there is a great deal of redundancy and a large value suggests a great deal of disorder. Entropy therefore is a measure of the information content of a message with respect to a particular model. It also determines the minimum number of bits.per symbol needed, on average, to represent a message generated by that model. Different coding models can produce different probability estimates. For a model that assigns a constant probability to each symbol of the alphabet, the entropy is calculated as

$$E = -\sum_{i=1}^{N} p_i \log_2 p_i \qquad \textbf{(1)}$$

where p_i is the probability of symbol I. Since the entropy of a message represents the lower bound of the number of bits required to losslessly represent the message, the only way to exceed this limit is to either change the probability generating model or to introduce loss into the message.

Chapter 2

Related Research

Information and Models

Even before the advent of computers there were attempts to minimize the alphabet used to represent a message. Morse code is an early example of a variable length coding scheme that reduced the code size for more common symbols. Reducing the symbol set from a fixed representation to a variable representation permits information to be encoded more efficiently. More recently, there has been a wealth of research on the reduction of message representation requirements [1,4,13,12,14,15,16,22,26,34,37].

In 1948, C. E. Shannon [29] and R. M. Fano independently developed a coding technique that attempted to minimize the number of bits used to encode a message when the probabilities of symbols in the message were known. Shortly afterward, D. A. Huffman [15] developed a technique that superseded Shannon-Fano coding and produces provably optimum code sets, resulting in marginally better performance than Shannon-Fano codes.

Huffman code is less than optimal when the probabilities of the symbol are not powers of two.

Another variation on the Huffman coding is *run-length encoding*, where the count of the run of a particular symbol is included in coding the scheme. If the symbol is optimized based on the number of unique runs and the runs are relatively long, this process can also reduce the number of symbols required for coding a message. This is particularly well suited for binary images since there are only two symbols and the runs can be very long. As the number of symbols increases and the runs become very short, this is a less efficient technique.

In 1987, a new more efficient method was introduced by I. H. Witten, R. M. Neal, and J. G. Cleary [34]. This method called *arithmetic coding,* works by assigning codes to symbols that have a known probability distribution and takes an entire message and encodes it as a single floating point number less than 1 and greater than or equal to 0. It achieves an average code length arbitrarily close to the entropy of the message. Arithmetic coding can more efficiently encode texts or images by eliminating the quantization effects of other coding techniques.

These encoding techniques are lossless and only reduce the space required by minimizing the representation alphabet. As end techniques, they can also be fed probabilities from another model. These are more sophisticated models that convert the original data into

another representation that improves compression. We will look at some probability models and how they generate the statistics that are fed to the encoder.

There are two approaches to image compression. Techniques based on exploitation of predictive means of an image are called *"spatial coding"* techniques. The first group of techniques exploits the redundancy in the data and attempt to predict the image causally. For example, an image with one color is fully predictable once the first pixel is known. Conversely, a white noise image is totally unpredictable and every pixel has to be stored to reconstruct the image. These forms of compression can be viewed as the attempt to transform the original image array $\{u_{i,j}\}$, into another array $\{v_{i,j}\}$, which has no redundancy and such that $\{u_{i,j}\}$ can be uniquely determined from $\{v_{i,j}\}$. The raw data rate of $\{v_{i,j}\}$ then determines the data rate of $\{u_{i,j}\}$. Generally, this type of process results in compression but with an accompanying distortion in the reconstructed array $\{u'_{i,j}\}$.

The second approach achieves compression by transforming a given image into another array such that the maximum information is packed into a minimum number of samples. These are based on, what would be loosely called, a frequency domain process. This is known as *"transform coding"* and are related to the non-causal representation of signals.

Spatial Methods

Lets briefly consider the predictive coding techniques. The principal is very simple. Since the image source is assumed to be highly correlated, then for any given element its neighbors should be closely related in value. So, we can use the preceding values of neighbors from the same line or previous lines to predict the value of the present element. Given our expectation of a highly correlated image, the difference of the predicted value and the actual value should be very small. The only error introduced is in quantizing the difference signal. The weakness of the predictive methods is that they are very sensitive to variations in the input data statistics and if a coding error occurs, it propagates throughout the remainder of the message.

Pulse-code modulation (PCM) [4]. Standard pulse code modulation (PCM) encoding is a common technique for encoding audio data. Telephone conversations and audio CDs both use conventional PCM. PCM samples a waveform at uniform steps and encodes the level of the waveform. Acceptable quality images can be obtained from compression ratios of 2.6:1.

Differential pulse-code modulation (DPCM) [13]. DPCM does not encode the level as PCM does, it instead encodes the difference from the last sample. Adaptive DPCM (ADPCM) [12] takes that a step further, and modifies the coding of the difference depending

on the state of the waveform. This technique generates ratios about 2.5:1, but adaptive versions can obtain ratios as high as 3.5:1.

Interpolative coding [4,23]. Most interpolative coders are of the zero-order or first-order models, giving compression ratios of around 4:1. Higher order polynomials can be used, but the computational complexity generally offsets the gain in the compression ratios.

Bit-plane coding [4]. This technique uses each bit of a byte as a parallel map of ones and zeros, these are then run-length encoded. Run-length encoding encodes the symbol and then the count of the number of consecutive repetitions of the symbol. Compression ratios of 4:1 can be obtained without using any visual system considerations.

Prediction by Partial Matching (PPM) [4]. This is a technique based on a finite context model. PPM makes its prediction based on a dynamic Markov model of various orders, most commonly order-1 or order-2. The limiting factor is the trade off between additional gains made in compression versus the space requirement of this model. Each order i of the model requires N^i possible strings to be considered, where N is the number of symbols in the alphabet. Compression ratios of 4:1 can be achieved.

Dictionary-based methods [4]. These may be either static or adaptive. Macro substitution methods use a dictionary to compress data. A string of symbols is encoded as a pointer into a codebook or dictionary. A static dictionary will compress an entire stream

using the same dictionary. An adaptive method, such as one of the Ziv-Lempel [37] schemes, is continually modifying its dictionary. Compression ratios as high as 8:1 can be obtained.

Transform Methods

It is at this point we will leave text compression behind, since text compression requires exact reconstruction and the following methods are subject to finite arithmetic precision errors. An alternative to predictive coding is *transform coding* where the operation is a successive multiplication of elements by a set of *basis* vectors $\{u_1, u_2, ..., u_n,\}$. Basis vectors in this case allow any solution to expressed uniquely as a linear combination of $\{u_1, u_2, ..., u_n\}$ in $L^2(R)$ the space of all square integrable functions.

Karhunen-Loève transform (KLT): This is the best linear transformation method. It is obtained by determining the basis vector set which diagonalizes the specific data covariance matrix, and it results in a transform domain covariance matrix of uncorrelated components. This packs the maximum average energy into a minimal set of samples. Since there is no fast algorithm associated with it, it is not used because of the computational load required.

Fast transformations: These transforms take a set of data in the spatial domain and convert them into an identical representation in the frequency domain. An important difference between these transforms and the KLT is that these transforms do not depend on the input image statistics. The three important features of a suitable transform are its compressional efficiency, which relates to concentrating the energy at low frequency, ease of computation, and minimum mean square error. One of the key features of the fast transforms is that they can be computed with $N \log_2 N$ or fewer operations as compared to N^2 in the standard Fourier transform.

Discrete Fourier transform (DFT): The Fourier transform converts the signal from the time domain to the frequency domain. This new domain has basis functions that are cosines and sines. The drawbacks to the Fourier transform is that it poorly localizes components in time and the transform is represented by both a real and imaginary component. It is described by:

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-j2\pi nk/N}, \; k = 0,...,N-1 \qquad (2)$$

Discrete cosine transform (DCT)[32]: This has many of the same properties of the KLT and it can be thought of as consisting of the cosine portion (the real part) of the DFT. The DCT is used as the core of the Joint Photographic Experts Group (JPEG) standard. It is represented by:

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cos\left[\frac{k\pi(2n+1)}{2N}\right], \quad k = 0,\ldots,N-1 \qquad \textbf{(3)}$$

Discrete wavelet transform (DWT)[36]: In the wavelet domain the basis functions are more complicated than the sine and cosine functions of the DFT. These basis functions are called "mother functions" or "wavelets." Unlike the sines and cosines which define the Fourier transform, there is not a single unique set of wavelets. It is this last class of fast transform methods that we will use as the core of our compression method.

Chapter 3

Preliminaries

Transforms

In order for us to understand how the wavelet transforms work, we need to discuss how transforms work in general. The definition of a Nth order linear transform of a one dimensional sequence $x(n)$; $n = 0, 1, ..., N-1$ is given by:

$$y(k) = \sum_{n=0}^{N-1} x(n) f(k,n) \qquad \text{for } k = 0, 1, ..., N-1 \tag{4}$$

Where $f(k,n)$ is a forward transformation kernel, and $y(k)$ are the transform coefficients. The inverse transform that recovers the input sequence is

$$\hat{x}(n) = \sum_{k=0}^{N-1} y(k)\, g(k,n) \qquad \text{for } n = 0, 1, ..., N-1 \tag{5}$$

where $g(k,n)$ is the inverse transformation kernel. The vector $\hat{x}$ is related to x by

$$\hat{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{e} \tag{6}$$

when $\boldsymbol{e}$, the error term, is 0 then the reconstruction is perfect. For the moment, we will require perfect reconstruction and consider $\boldsymbol{e}$ to be 0 and $\hat{\boldsymbol{x}}$ to be equal to $\boldsymbol{x}$.

In matrix notation

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{F}\boldsymbol{x} \\ \boldsymbol{x} &= \boldsymbol{G}\boldsymbol{y} \\ \boldsymbol{G} &= \boldsymbol{F}^{-1} \end{aligned} \tag{7}$$

for input vector $\boldsymbol{x}$, and coefficient vector $\boldsymbol{y}$ and

$$\begin{aligned} \boldsymbol{F} &= \{f(m,n)\}_{m,n=0,1,\ldots,N-1} \\ \boldsymbol{G} &= \{g(m,n)\}_{m,n=0,1,\ldots,N-1} \end{aligned} \tag{8}$$

we will use the notation

$$\begin{aligned} \boldsymbol{G} &= \{\boldsymbol{g}_k\}_{k=0,1,\ldots,N-1} \\ \boldsymbol{g}_k &= \{g(m,k)\}_{m=0,1,\ldots N-1} \end{aligned} \tag{9}$$

where $\boldsymbol{g}_k$ are basis vectors. Therefore

$$\boldsymbol{x} = \boldsymbol{G}\boldsymbol{y} = \sum_{k=0}^{N-1} y(k)\boldsymbol{g}_k \tag{10}$$

and $\boldsymbol{x}$ is the weighted sum of basis vectors, where the weights are the values of the transform coefficients.

Let $\boldsymbol{F}$ be real, the class of *orthogonal* transforms is defined by

$$\boldsymbol{F}^{-1} = \boldsymbol{F}^{T} \tag{11}$$

implying

$$\boldsymbol{F}^{T}\boldsymbol{F} = \boldsymbol{F}\boldsymbol{F}^{T} = \boldsymbol{I} \tag{12}$$

where $\boldsymbol{I}$ is the identity matrix of order N. A real matrix is orthogonal if and only if its columns and rows form an orthonormal set where every vector has unit length, i.e.

$$f_i^{T} \cdot f_j = \delta_{ij} = \begin{cases} 0 \text{ for } i \neq j \\ 1 \text{ for } i = j \end{cases} \tag{13}$$

where $\delta_{i,j}$ is the *Kronecker delta* function and therefore the inverse transform is just the transpose of $\boldsymbol{F}$:

$$\boldsymbol{G} = (\boldsymbol{F}^{-1})^{T} \tag{14}$$

Orthogonality is a necessary property for basis vectors that are used to decompose an input into uncorrelated components in an N-dimensional space. *Orthonormality* is an even stronger property; it leads to transforms where the average sum of the variances of the elements of the output $\boldsymbol{y}$ are equal to σ_x^2, the variance of the elements of $\boldsymbol{x}$. This implies that

whatever the average reconstruction error variance is, it is equal the error variance introduced in the quantization of the transform coefficients. While it is true that finite precision of the calculations can introduce errors also, we will consider them insignificant.

The two-dimensional transform can be generalized from **(4)** and **(5)**

$$y(k,l) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m,n)\ f(k,l,m,n) \tag{15}$$

$$x(m,n) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} y(k,l)\ g(k,l,m,n) \tag{16}$$

each of the above describe a N^2 element sequence with transformation filters of $f(k,l,m,n)$ and $g(k,l,m,n)$ respectively. For a 2-dimensional image transform we will generally assume a $N \times N$ image and that the transform filters are *separable*. Separable filters allow horizontal and vertical operations to be done separately and as such the filters can be decomposed into

$$\begin{aligned} f(k,l,m,n) &= f_v(k,m)\ f_h(l,n) \\ g(k,l,m,n) &= g_v(k,m)\ g_h(l,n) \end{aligned} \tag{17}$$

This permits us to perform the transform in two one-dimensional operations. The two dimensional transform to obtain y can now be done as

$$y(k,l) = \sum_{m=0}^{N-1} f_v(k,m) \sum_{n=0}^{N-1} x(m,n)\ f_h(l,n)$$
$$= \sum_{m=0}^{N-1} f_v(k,m)\ y(m,l) \qquad \textbf{(18)}$$

Defining $\boldsymbol{F}_v$ and $\boldsymbol{F}_h$ as arrays of the appropriately shifted filters $\boldsymbol{f}_v$ and $\boldsymbol{f}_h$ and requiring the filters to be *symmetrical* allows us to establish

$$\boldsymbol{F}_v = \boldsymbol{F}_h = \boldsymbol{F} \qquad \textbf{(19)}$$

Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ represent the arrays containing the elements $x(m,n)$ and $y(k,l)$ respectively, and we can further say

$$\boldsymbol{Y} = \boldsymbol{F}\,\boldsymbol{X}\,\boldsymbol{F}^{T} \qquad \textbf{(20)}$$

and with $\boldsymbol{F}^{-1} = \boldsymbol{F}^{T}$,

$$\boldsymbol{X} = \boldsymbol{F}^{T}\,\boldsymbol{Y}\,\boldsymbol{F} \qquad \textbf{(21)}$$

Observe that $\boldsymbol{F}$ is simply the one-dimensional transform and that this operation is only possible with separable filters.

Sub-band Transforms

Now with the understanding of how the transform operates, we need to introduce the concept of *sub-band* transforms. These transforms convert the original signal into two or more sub-bands. We will only be concerned with band splitting or two bands, although the process can be applied to multiple sub-bands.

In general, the band splitting process involves *convolving*, or applying the bandpass transform to the input elements, and then decimating the resulting output. *Decimation*, also known as sub-sampling, samples and retains only every *k*th element of the output, in our case we will use a two-channel sub-band transform and sub-sample by two. This means that we will use a pair of specially designed filters, a high bandpass filter $\boldsymbol{G}$ and low bandpass filter $\boldsymbol{H}$. Convolving and decimating the input results in a sub-band output of $N/2$ elements for a N element input, so there is a lowpass output of $N/2$ elements and a high-pass output of $N/2$ elements.

To illustrate, using the z transform

$$x(n) \rightarrow x(z) = \sum_{n=0}^{N-1} x(n)\, z^{-n} \qquad \textbf{(22)}$$

suppose we have a signal $u(0), u(1), u(2), \ldots, u(2N-1)$ that we wish to decimate or sub-sample

$$v(n) = u(2n) = \frac{1}{2}(u(n) + (-1)^n u(n)) \tag{23}$$

then we can say

$$\begin{aligned} v(z) &= \sum_{n=0}^{N-1} v(n) z^{-n} \\ &= v(0)z^{-0} + v(1)z^{-1} + \ldots + v(n)z^{-(n-1)} \end{aligned} \tag{24}$$

if

$$\begin{aligned} \frac{1}{2}\left(u(n) + (-1)^n u(n)\right) &= v(0)z^{-0} + 0 \cdot z^{1/2^{-1}} + v(1)z^{1/2^{-1}} + \ldots \\ &\quad + v(N-1)z^{1/2^{-(2N-1)}} + 0 \cdot v(z)^{1/2^{-(2N-1)}} \\ &= \frac{1}{2}\left(u(0) + (-1)^0 u(0)\right)z^{-0} + \frac{1}{2}\left(u(1) + (-1)^1 u(1)\right)z^{1/2^{-1}} + \ldots \\ &\quad + \frac{1}{2}\left(u(2N-1) + (-1)^{2N-1} u(0)\right)z^{1/2^{-(2N-1)}} \\ &= \frac{1}{2}\left[u(0)\left(z^{1/2}\right)^{-0} + u(1)\left(z^{1/2}\right)^{-1} + \ldots\right] \\ &\quad + \frac{1}{2}\left[(-1)^0 u(0)z^{-0} + (-1)^1 u(1)\left(z^{1/2}\right)^{-1} + \ldots\right] \\ &= \frac{1}{2}\left[u(z^{1/2}) + u(-z^{1/2})\right] \\ &= v(z) \end{aligned} \tag{25}$$

Therefore, every other sample is used during sub-sampling. The interpolation or up-sampling is similar in approach, using $v(0), v(1), v(2), \ldots, v(N-1)$ the sampled signal we want to recreate $u(0), u(1), u(2), \ldots, u(2N-1)$

To reconstruct from the transformed elements, we must first *up-sample* or *interpolate*, we do this by inserting zeros between the N samples, such that

$$\begin{aligned} u(0) &= v(0) \\ u(1) &= 0 \\ u(2) &= v(1) \\ u(3) &= 0 \\ &\vdots \\ u(2N-2) &= v(N-1) \\ u(2N-1) &= 0 \end{aligned} \tag{26}$$

this is done by

$$\begin{aligned} u(n) = \sum_{n=0}^{2n-1} v(n) z^{-n} &= v(0) + 0(z^{-1}) + v(1)z^{-2} + (0)z^{-3} + \ldots + v(N-1)z^{2N-1} \\ &= v(0) + v(1)z^{-2} + \ldots + v(N-1)z^{2N-1} \\ &= v(0) + v(1)z^{2^{-2}} + \ldots + v(N-1)z^{2^{2N-1}} \\ &= v(z^2) \end{aligned} \tag{27}$$

We can then convolve the high and low pass samples with reconstruction filters $\boldsymbol{G}'$ and $\boldsymbol{H}'$ respectively, and then add the two signals together to obtain the original input. The forward application stage of the sub-band transforms is called *analysis* and the inverse application is called *synthesis* as shown in **Figure 1**.

The process of sequentially reprocessing the outputs of the analysis stage by another application of the analysis stage is known as a *uniform* cascade system, when the analysis

stage is only reapplied to the low-pass output of the analysis, it is known as a non-uniform or *pyramid* cascading system.

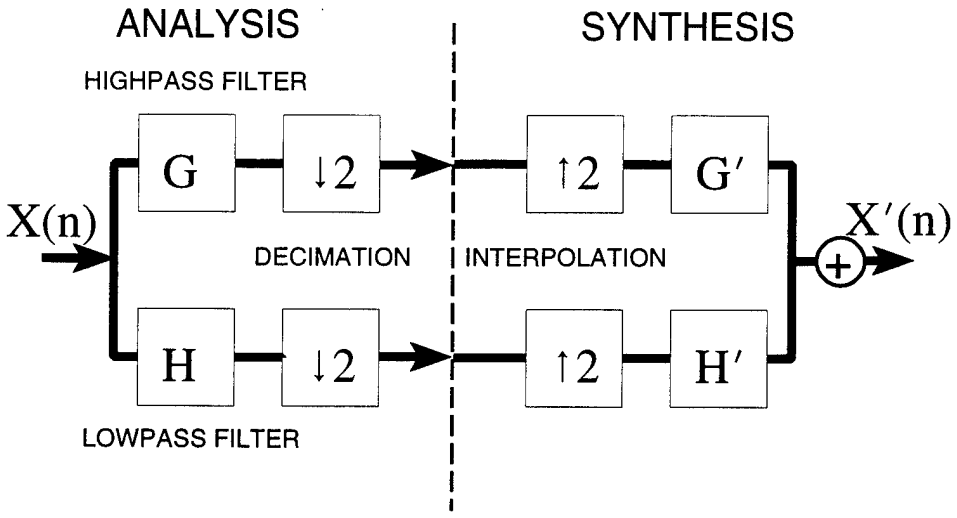

Figure 1: The two-channel band-pass transform operation.

What are wavelets?

Wavelets are a family of functions that satisfy certain requirements. The name *wavelet* should bring to mind a function that waves above and below the x-axis and should therfore integrate to zero. Another connotation of *wavelet* suggests that it should be small, or in other words the function has to be well localized. Other constraints on the functions insure quick and easy calculation of the direct and inverse wavelet transform.

There are a broad variety of wavelets, from smooth wavelets, compactly supported wavelets, wavelets with simple mathematical expressions, wavelets with simple associated filters, etc. The most simple is the *Haar wavelet,* and we discuss it as an introductory example. Like sines and cosines in Fourier analysis, wavelets are used as basis functions in representing other functions. Once the wavelet (sometimes called *the mother wavelet)* $\psi(x)$ is fixed, a basis can be made of translations and dilations of the mother wavelet $\psi\left(\frac{x-b}{a}\right)$, $(a,b) \in R^{+} \times R$. It is convenient to take special values for a and b in defining the wavelet basis: $a = 2^{-j}$ and $b = k \cdot 2^{j}$, where k and j are integers. This choice of a and b will give a sparse basis. This choice also connects multiresolution analysis in signal processing with the world of wavelets.

Why not use the traditional Fourier methods? The Fourier transform has been used for many years in signal analysis. It has proven invaluable in applications ranging from pattern recognition to image processing. There are certain undesirable limitations that are inherent in it though, and the wavelet transform has the same power and versatility, but without the limitations.

The Fourier basis functions are localized in frequency but not in time. Small frequency changes in the Fourier transform will produce changes everywhere in the time domain. In contrast, wavelets are local in both frequency/scale (via dilations) and in time (via translations).

The Fourier transform works on the assumption that the original time domain function (i.e., signal or image) is periodic in nature. As a result, the Fourier transform has difficulty with transient components of the image that are localized in time. In other words, it poorly localizes sharp transitions in the image such as edges. The Fourier transform of a signal does not convey any information about the translation of the signal in time. The only way to handle this shortcoming is to window the input data so that the sampled data converges to zero at the endpoints.

Wavelet transforms provide a more compact representation of many classes of functions. For example, functions with discontinuities and functions with sharp spikes usually take substantially fewer wavelet basis functions than sine-cosine basis functions to achieve a comparable approximation. This sparse coding makes wavelets excellent tools in image compression.

In recent years, new families of orthonormal basis functions have been developed to overcome these shortcomings. This family of functions are called "*wavelets*." Unlike the sine and cosine.wave of the Fourier transform and discrete cosine transform, they need not have infinite duration. They can be non-zero for only a small range of the function. This short duration is referred to as "compact support" and allows a time domain function to be translated into a representation that is localized in frequency and time. This is the behavior that has provided new advances in signal processing.

Large and noisy data sets can be easily and quickly transformed by the DWT. The data are coded by the wavelet coefficients. In addition, the DWT is much faster than the DFT. It is well known that the computational complexity of the DFT is $O(n \log_2 n)$ [25]. For the discrete wavelet transform the computational complexity goes down to $O(n)$.

How do the wavelets work?

The Haar wavelet

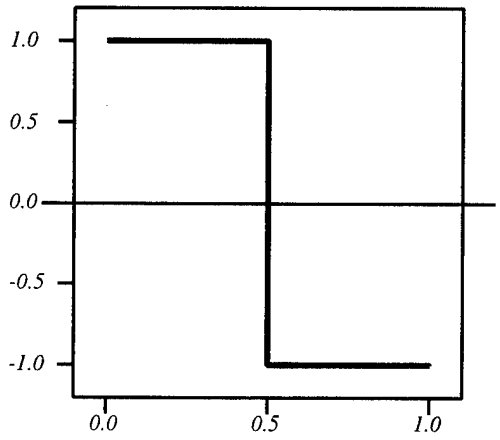

Figure 2: Haar Wavelet

To explain how wavelets work, we start with an example. We choose the simplest and best known of all wavelets, the Haar wavelet, $\psi(x)$. The Haar wavelet has been known for more than eighty years and has been used in various mathematical fields. It is known that any continuous function can be approximated uniformly by Haar functions. It is a step

function taking values 1 and -1, on [0,1/2) and [1/2,1), respectively. The graph of the Haar wavelet is given in **Figure 2**.

Dilations and translations of the function ψ,

$$\psi_{jk}(x) = \text{const} \cdot \psi(2^j x - k) \tag{28}$$

where j is the *dilation* function and k is the *translation* function, define an orthogonal basis in $L^2(R)$ (the space of all square integrable functions). This means that any element in $L^2(R)$ may be represented as a linear combination (possibly infinite) of these basis functions.

The orthogonality of ψ_{jk} is easy to check. It is apparent that

$$\int \psi_{jk}(x) \cdot \psi_{j'k'}(x) = 0 \tag{29}$$

whenever $j = j'$ and $k = k'$ is not satisfied simultaneously. If j ≠ j' (say $j' < j$), then nonzero values of the wavelet $\psi_{kj'}$, are contained in the set where the wavelet $\psi_{kj'}$ is constant. That makes integral **(29)** equal to zero.

If $j = j'$, but k ≠ k', then at least one factor in the product $\psi_{kj'} \bullet \psi_{kj}$ is zero. Thus the functions ψ_{kj} are orthogonal. The constant that makes this orthogonal basis orthonormal is $2^{j/2}$. Indeed, from the definition of norm in L^2:

$$1 = (\text{const})^2 \int \psi^2(2^j x - k)dx = (\text{const})^2 \cdot 2^{-j} \int \psi^2(t)dt = (\text{const})^2 \cdot 2^{-j} \tag{30}$$

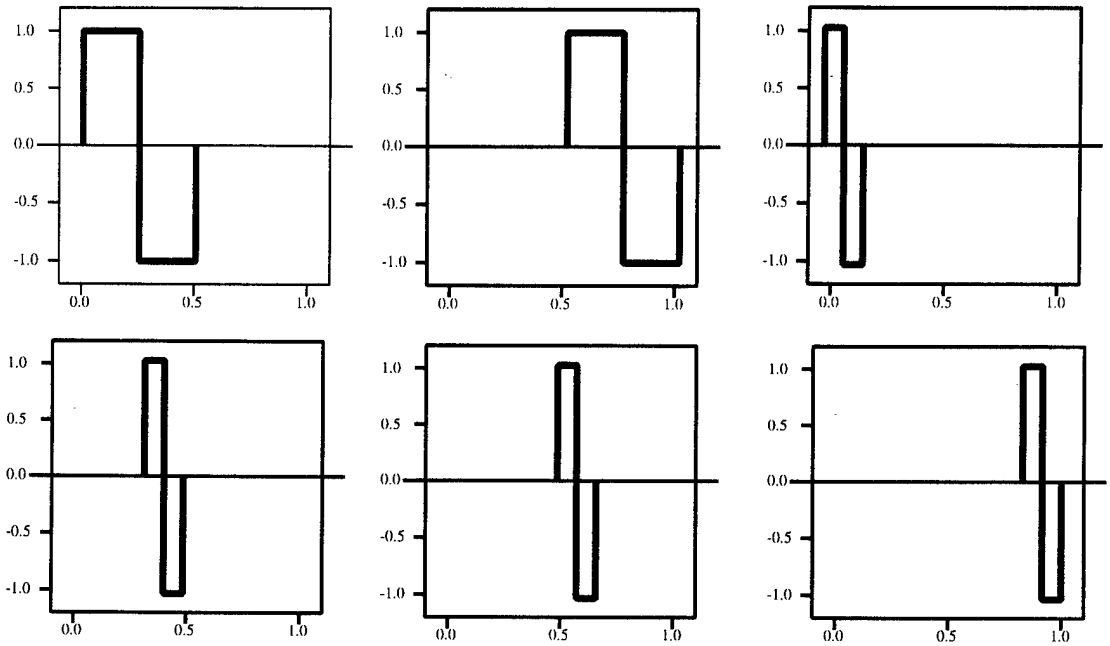

Figure 3: Dilations and translations $\psi_{j,k}$ of the Haar wavelet on [0,1]

The functions ψ_{10}, ψ_{11}, ψ_{20}, ψ_{21}, ψ_{22}, ψ_{23} in are shown in **Figure 3**. The set $\{\psi_{jk},\ j\in Z,\ k\in Z\}$ defines an orthonormal basis for L^2. Alternatively we will consider orthonormal bases of the form $\{\phi_{j_0,k},\ \psi_{jk},\ j\geq j_0,\ k\in Z\}$, where ϕ_{00} is called the *scaling function* associated with the wavelet basis ϕ_{jk}. The set $\{\phi_{j_0,k},\ k\in Z\}$ spans the same subspace as $\{\phi_{j,k},\ \psi_{jk},\ j<j_0,\ k\in Z\}$. We will later define ϕ_k. For the Haar wavelet basis the scaling function is very simple. It is unity on the interval [0,1), i.e.

$$\phi(x) = 1(0 \leq x \leq 1) \tag{31}$$

Let $y = (y_0, y_1, \ldots, y_{2^n-1})$ be the data vector of size 2^n. The data vector can be associated with a piecewise constant function f on [0,1) generated by y as follows,

$$f(x) = \sum_{k=0}^{2^n-1} y_k \cdot 1(k2^{-n} \le x \le (k+1)2^{-n}) \qquad \textbf{(32)}$$

The (data) function f is obviously in the $L^2[0, 1)$ space, and the wavelet decomposition of f has the form:

$$f(x) = c_{00}\phi(x) + \sum_{j=0}^{n-1} \sum_{k=0}^{s^j-1} d_{jk}\psi_{jk}(x) \qquad \textbf{(33)}$$

The sum with respect to j is finite because f is a step function, and everything can be exactly described by resolutions up to the $(n - 1)$-st level. For each level the sum with respect to k is also finite because the domain of f is finite. In particular, no translations of the scaling function ϕ_{00} are required.

An obvious disadvantage of the Haar wavelet is that it is not continuous, and therefore it is not the best choice for representing smooth functions. Therefore, we need to extend this concept to additional wavelet representations.

What are the requirements for a filter?

Daubechies discovered that the wavelet transform can be implemented with a specially designed pair of *finite impulse response (FIR)* filters called a *quadrature mirror filter (QMF)*. A FIR performs the dot or sum of products between the filter coefficients and the discrete

samples in the tapped delay line of the filter. Passing a set of discrete samples through a FIR filter is a discrete convolution of the signal with the filter's coefficients.

To picture this process we consider a four coefficient filter designed by Daubechies. The transform matrix for this filter is

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & 0 & \dots & 0 & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & 0 & \dots & 0 & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & & & \ddots & & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & 0 & 0 & \dots & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & 0 & \dots & 0 & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & 0 & \dots & 0 & 0 & 0 & c_3 & -c_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{n-3} \\ u_{n-2} \\ u_{n-1} \\ u_n \end{bmatrix} \quad \textbf{(34)}$$

There are several points to note about this matrix. The odd rows are copies of the first row shifted by two each time. The even rows are not the same filter, but do use the same coefficients, therefore the even rows actually perform a different convolution. Note that the filter wraps around the matrix boundary. The overall effect of the matrix is to do two separate convolutions and to decimate the results by two. The results are interleaved with each other. The filter c_0, c_1, c_2, c_3, is an averaging or smoothing filter generally called H and the filter c_3, $-c_2$, c_1, $-c_0$, does not average because of the minus signs and therefore is the detail filter called G.

There are some requirements that must be placed on the filter to make them perform properly. First, the c's must be chosen so that the detail filter G will produce a zero response to a sufficiently smooth signal. In the case of a four coefficient filter this means that we must have vanishing moments of order two. Another condition is required to reconstruct the original input from the output is that the matrix must be orthogonal. Hence, the reconstruction matrix for equation **(34)** is

$$\begin{bmatrix} c_0 & c_3 & 0 & 0 & 0 & \dots & 0 & 0 & c_2 & c_1 \\ c_1 & -c_2 & 0 & 0 & 0 & \dots & 0 & 0 & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & 0 & \dots & 0 & 0 & 0 & 0 \\ c_3 & -c_0 & c_1 & -c_2 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & & & \ddots & & & & & \\ 0 & 0 & \dots & 0 & c_2 & c_1 & c_0 & c_3 & 0 & 0 \\ 0 & 0 & \dots & 0 & c_3 & -c_0 & c_1 & -c_2 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & c_2 & c_1 & c_0 & c_3 \\ 0 & 0 & \dots & 0 & 0 & 0 & c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_{n-3} \\ v_{n-2} \\ v_{n-1} \\ v_n \end{bmatrix} \tag{35}$$

The only way that **(34)** and **(35)** can be orthogonal is if the following equations are satisfied

$$\begin{aligned} c_0^2 + c_1^2 + c_3^2 + c_3^2 &= 1 \\ c_0 c_2 + c_1 c_3 &= 0 \end{aligned} \tag{36}$$

Additionally, for the vanishing moments to be satisfied requires that

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0(c_3) - 1(c_2) + 2(c_1) - 3(c_0) &= 0 \end{aligned} \tag{37}$$

also be met. The equation sets in **(36)** and **(37)** were first solved by Daubechies [7] and are listed as coefficient set 4 in **Table 1**. As the number of coefficients increase by two, the number of orthogonality requirements increase by one.

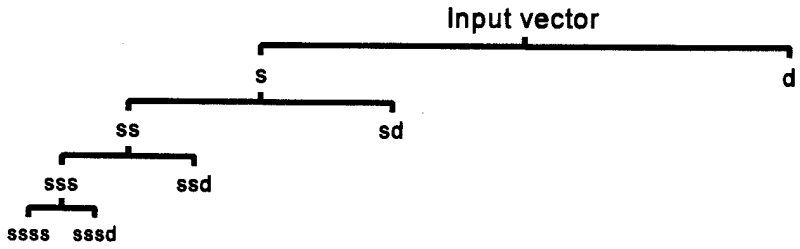

Figure 4: Pyramid output of the wavelet transform

QMF filters are unique because the frequency responses of the two filters separate the high and the low frequency components of the input data. A dividing point is generally found between 0 and half the data sampling (Nyquist) frequency. The outputs of the QMF filter pair are decimated by two, in other words every other sample is discarded, see **Figure 1**. The high frequency pass resolves the details of the image and the low frequency pass resolves the averages (smoothness) of the image. The lowpass portion of the data set can then be repeatedly fed to the QMF. **Figure 4** demonstrates how the input signal is divided

into a smooth (s) and detail (d) component. The smooth component is recursively processed to create new smooth and detail components thereby forming a pyramid.

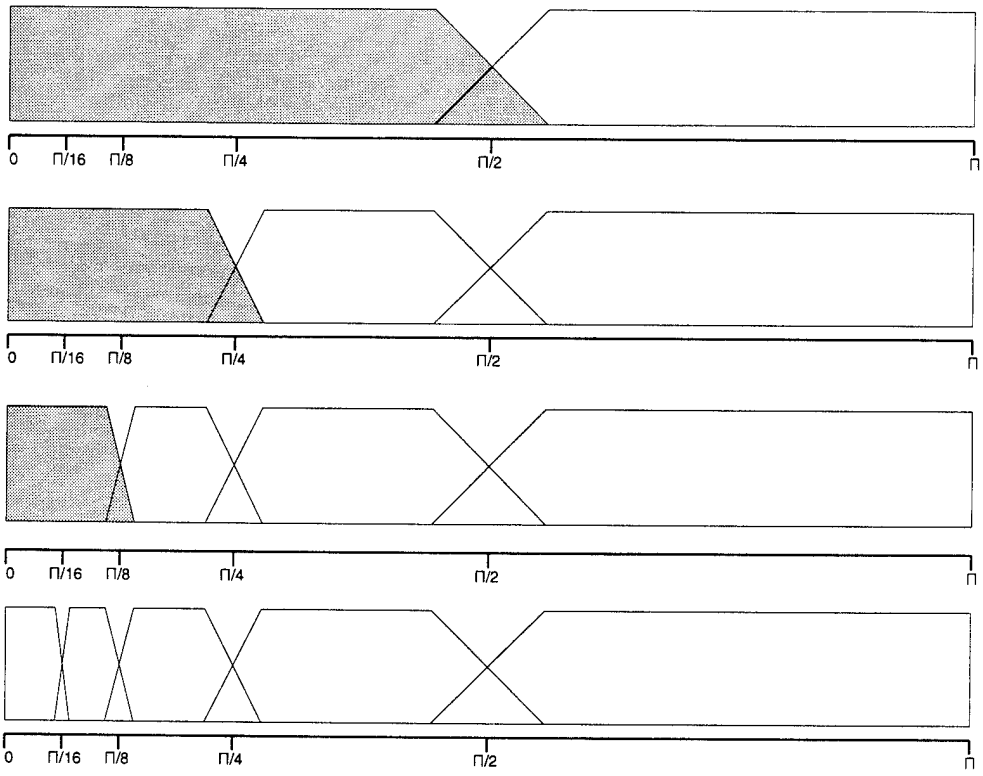

Figure 5: Spectral octave bands.

The recursive pyramiding of an image divides the spectrum of the original signal into octaves bands with successively coarser measurements in time as the width of each spectral band decreases in frequency, **Figure 5**. This has the effect of folding most of the energy contained in the full vector across the Nyquist frequency into the lower half of the vector as each level is processed. The grey band reflects the vector processed for the next level of processing.

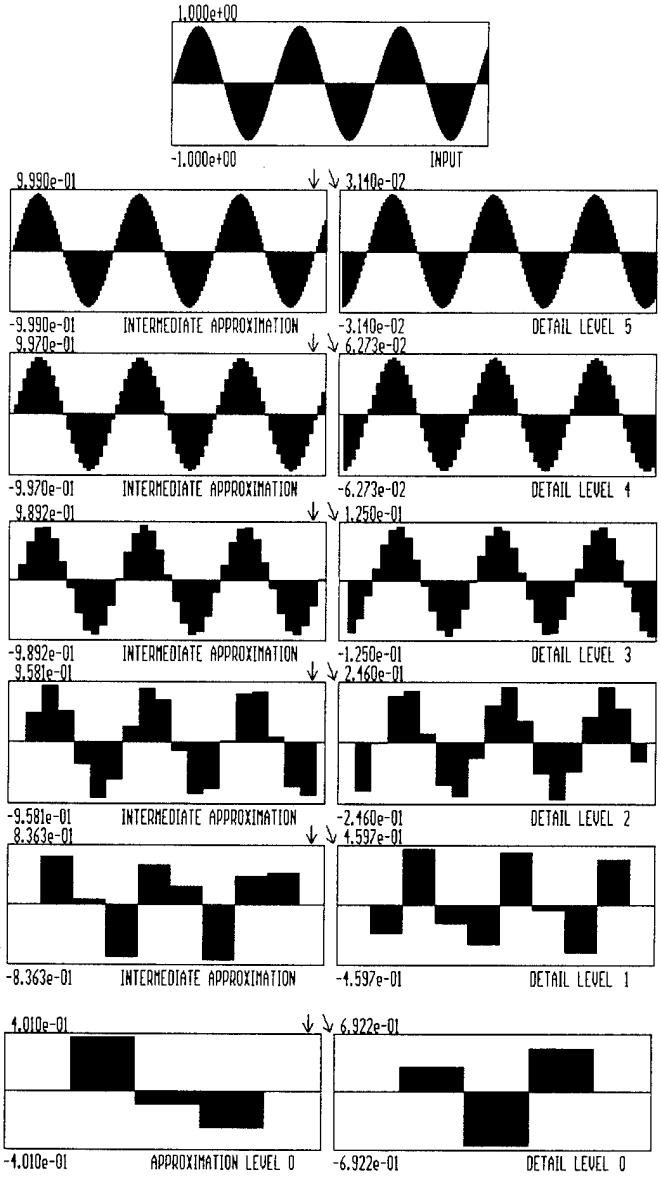

Figure 6: Haar wavelet approximation of a sine wave at six levels of resolution.

As an example of the process and the output, a 128-point sine wave is decomposed by the Haar basis set in **Figure 6** each level shows an approximation reconstruction for that level. **Figure 7** shows the actual signal data set superimposed with the transform

coefficients. Note how the amplitudes of the transform coefficients increase as the resolution decreases.

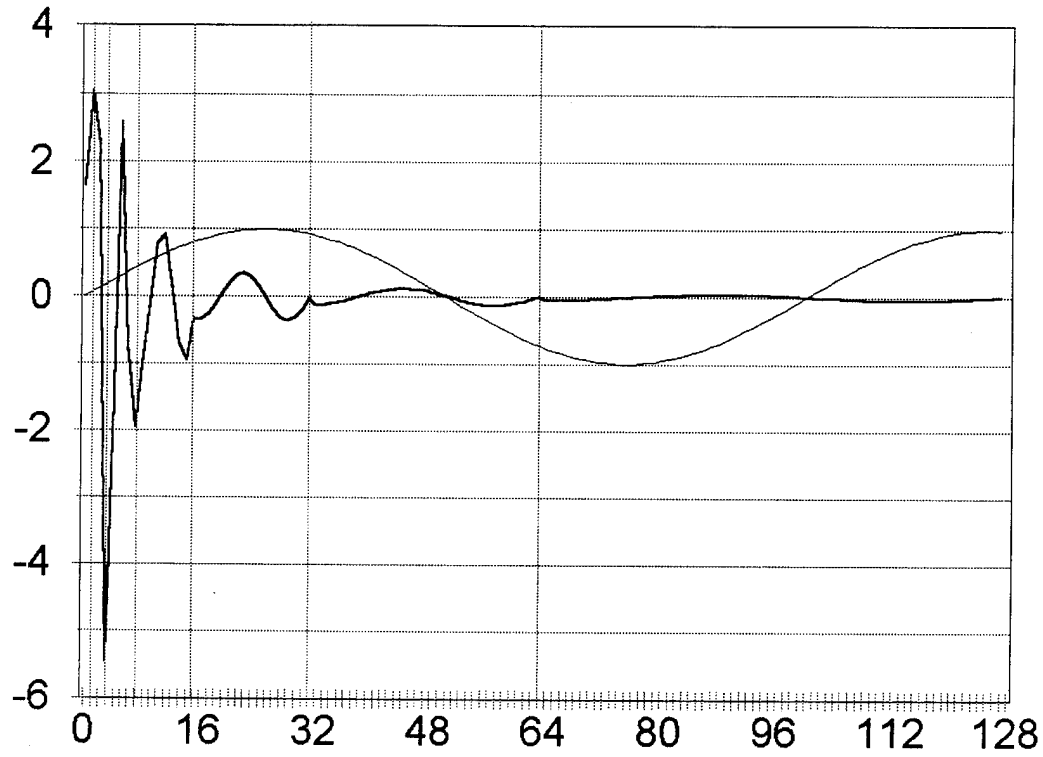

Figure 7: 128 point sine wave decomposition with Haar basis set.

Mallat [161] has shown that the pyramid algorithm can be applied to the wavelet transform by using the wavelet coefficients as the filter coefficients of the QMF filter pairs. The same wavelet coefficients are used in both the high-pass and lowpass filters. The lowpass coefficients are associated with a_k of the scaling function ϕ.

$$\phi(t) = \sum_{k \in Z} a_k \phi(2t-k) \tag{38}$$

The output of each lowpass filter is the approximation components for that level of the tree. The high-pass filter is associated with the a_k of the wavelet function ψ (note the alternating sign change).

$$\psi(t) = \sum_{k \in Z} (-1^k) a_{k+1} \phi(2t+k) \tag{39}$$

The output of the high-pass filter is the detail components of the original signal at resolution 2^j, where j is the level of the pyramid. The lowpass output of the previous level is used to generate a new set of high-pass and lowpass outputs for the next level of the tree. Decimation by two corresponds to the multiresolutional nature (the j parameter) of the scaling and wavelet functions.

What do wavelets look like?

The following are visualizations and listings of the Daubechies filter set. The filter visualizations were created with a single impulse of 1000.0 in the sixth component a 128-point data set in the frequency domain. This single impulse was then returned to the time domain, much in the same fashion that pure sine waves are generated from a single impulse in the frequency domain and then returned to the time domain with the Fourier transform.

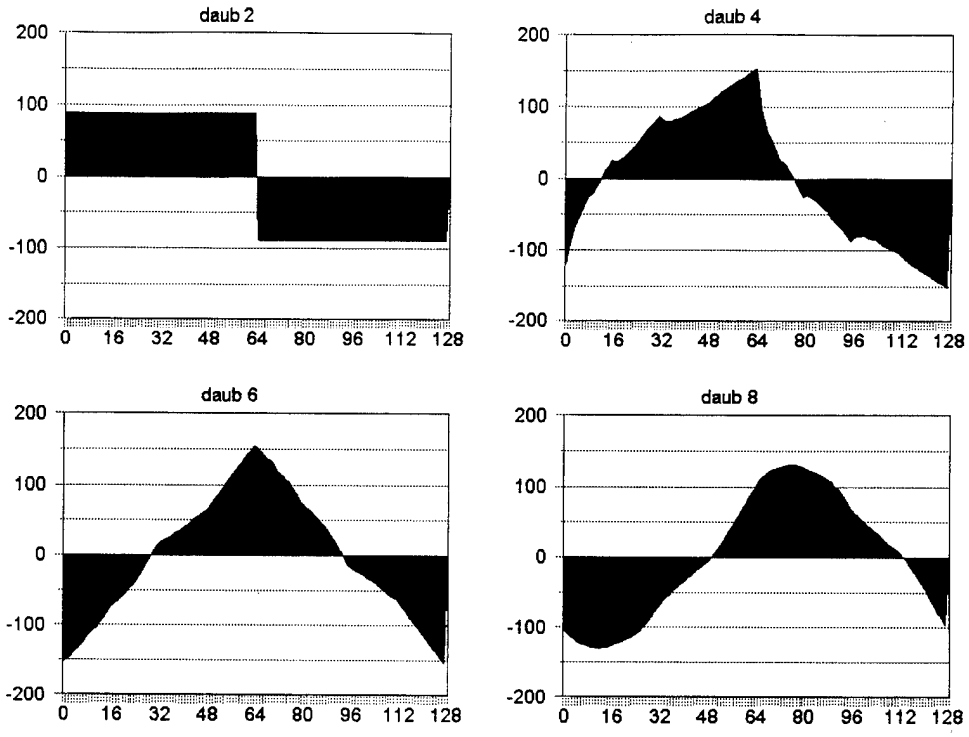

Figure 8: Daubechies' wavelet bases 2-8.

	d2	d4	d6	d8	d10
1	0.707106781187	0.482962913145	0.332670552950	0.230377813309	0.160102397974
2	0.707106781187	0.836516303738	0.806891509311	0.714846570553	0.603829269797
3		0.224143868042	0.459877502118	0.630880767930	0.724308528438
4		-0.129409522551	-0.135011020010	-0.027983769417	0.138428145901
5			-0.085441273882	-0.187034811719	-0.242294887066
6			0.035226291882	0.030841381836	-0.032244869585
7				0.032883011667	0.077571493840
8				-0.010597401785	-0.006241490213
9					-0.012580751999
10					0.003335725285

Table 1: Daubechies' wavelet coefficients 2-10.

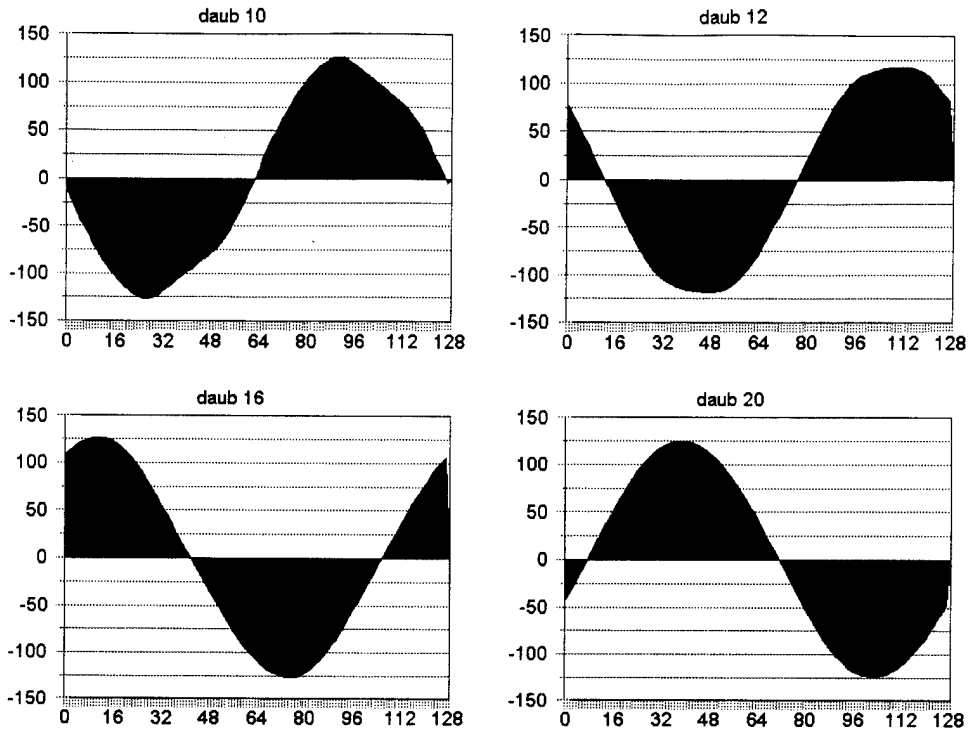

Figure 9: Daubechies' wavelet bases 10, 12, 16 and 20.

	d12	d14	d16	d18	d20
1	0.111540743350	0.077852054085	0.054415842243	0.038077947364	0.026670057901
2	0.494623890398	0.396539319482	0.312871590914	0.243834674613	0.188176800078
3	0.751133908021	0.729132090846	0.675630736297	0.604823123690	0.527201188932
4	0.315250351709	0.469782287405	0.585354683654	0.657288078051	0.688459039454
5	-0.226264693965	-0.143906003929	-0.015829105256	0.133197385825	0.281172343661
6	-0.129766867567	-0.224036184994	-0.284015542962	-0.293273783279	-0.249846424327
7	0.097501605587	0.071309219267	0.000472484574	-0.096840783223	-0.195946274377
8	0.027522865530	0.080612609151	0.128747426620	0.148540749338	0.127369340336
9	-0.031582039318	-0.038029936935	-0.017369301002	0.030725681479	0.093057364604
10	0.000553842201	-0.016574541631	-0.044088253931	-0.067632829061	-0.071394147166
11	0.004777257511	0.012550998556	0.013981027917	0.000250947115	-0.029457536822
12	-0.001077301085	0.000429577973	0.008746094047	0.022361662124	0.033212674059
13		-0.001801640704	-0.004870352993	-0.004723204758	0.003606553567
14		0.000353713800	-0.000391740373	-0.004281503682	-0.010733175483
15			0.000675449406	0.001847646883	0.001395351747
16			-0.000117476784	0.000230385764	0.001992405295
17				-0.000251963189	-0.000685856695
18				0.000039347320	-0.000116466855
19					0.000093588670
20					-0.000013264203

Table 2: Daubechies' wavelet coefficients 12-20.

Quantization

Quantization is the process of converting a given symbol value into a value taken from a set of finite possibilities. Up to this point all of the methods discussed are lossless except for finite precision arithmetic errors. It is normally the step in a compression scheme that introduces the noise (loss) into the system. Consider as a form of quantization, the conversion from real numbers to integers, the integers are finite for a given range and the loss of the fractional part of the real number is the noise that is introduced. Quantization allows the number of symbols to be reduced and correspondingly improves encoding efficiency.

After using the DWT transform on an image the resulting array will be in floating point representation. We have quadrupled the size of the image since the floating point representation requires in general four times the storage (four bytes output versus one byte input). Now, even to represent the image in the same space as the original will require conversion to a finite precision representation. It is at this point that we begin to introduce noise into the image and a corresponding loss of fidelity.

First, let us assume that the output range of our transform can be truncated or rounded to fit into an integer representation. Using our model from before, the quantization step is inserted between transform steps as in **Figure 10**.

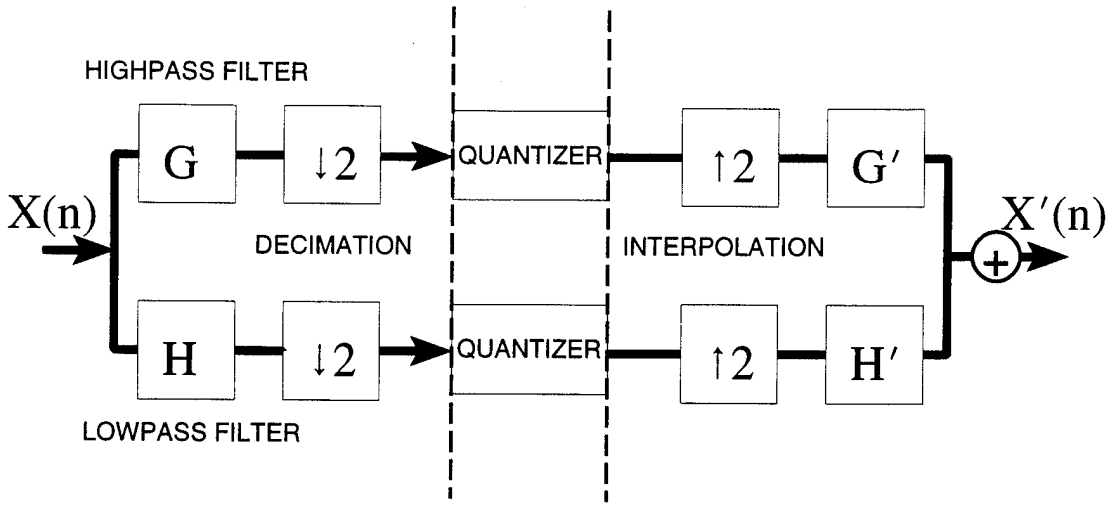

Figure 10: Transform model with quantization step.

One of the key features of the wavelet transform is that the coefficients can be severely truncated and still reconstruct an adequate representation of the input. The next step in this process then is to truncate or threshold the data so that a minimal set is retained.

Maximizing Retained Data

It is in this portion of the scheme that we wish to address the problem of determining which coefficients we wish to keep to represent the image for an expected byte budget. Recall for a moment that the transform removes the smoothness of the image at each level of the transform and its inability to absorb all of the changes in the image are the coefficients that are left behind in the high pass portion of the sample set. If the details are small enough

then they can be omitted without significantly affecting the overall quality of the reconstructed data.

This omission of small details is called *thresholding*. In general, one of two methods are used, *hard thresholding* is when any coefficient value below an established threshold is reduced to zero. Soft thresholding is a shrinking of values towards zero, any coefficient below a certain value is reduced to zero and all other values are shifted towards zero by the threshold amount. The threshold can be set by a fixed value or by a percentile value based on the total range of the coefficients. For example, if we wish to reduce the number of coefficients by half then we set the threshold at the median value. It is thresholding along with the finite representation from quantization that will allow us to further compress our image.

From our discussion of the transform, each level of depth of the transform is at a coarser resolution by a factor of two. The detail coefficients in the highband pass of a given level are only correlated to the highband pass of the next level of the transform in the sense that if a level has an area of significant detail coefficient activity then it indicates that there may be more significant detail coefficients at a finer resolution level. Accordingly, if the coefficients are small then the detail probably contributes little to a relatively smooth area of the image. What this means is simply, if there are a lot of edges or variations in an area at a given resolution then it is a good indicator of additional detail at a finer scale.

Conversely, smoothly transitioning areas will not gain significant detail at greater levels of resolution. We will refer to this concept as *self-similarity*.

We will exploit the self-similarity between the various levels of the transformation. Although the transform removes most of the correlation of the image, it does not remove all of it and the correlation left is not necessarily predictable. It is this self-similar relationship between levels we will use to model our storage of the image. The concept is straight forward, use a binary tree to represent each level of the transformation with the parent pointing to the children that are the coefficients at the next level of higher resolution. This is similar to research done by J. M. Shapiro [30] except that they used a quadtree model with only a single decision level. If the value of the coefficient is significant, then keep it. If the value of the coefficient is not significant, nor are any of the values of the descendants, then discard it and prune the branch. Thus, **Figure 11** shows our modified compression scheme model.

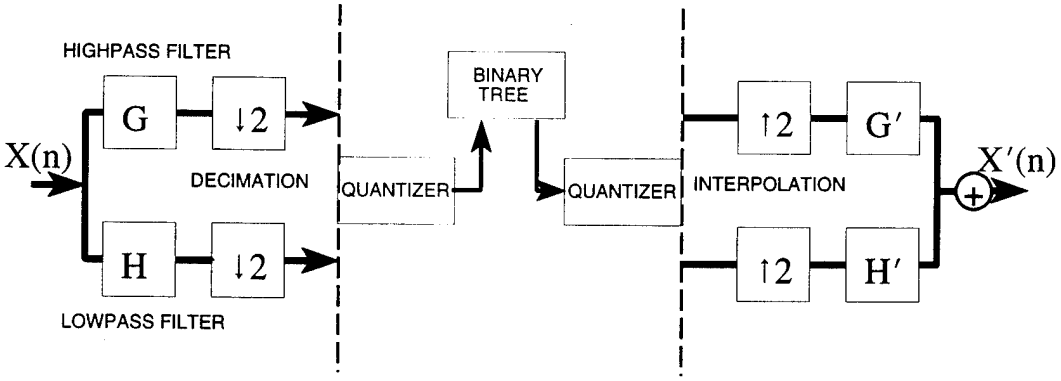

Figure 11: Model with tree storage step.

Initially, the problem is how to build the decision process into the tree structure. Special symbols can be allocated from the available alphabet to direct the construction and reconstruction of the binary tree. Alternatively, a bit prefix can be used from the bits allocated for the numerical representation of the coefficient.

We will evaluate these models on a simple example using a four-level transform of 31 elements. Placing the root at the coarsest level (deepest level of the transformation), we can imagine that the same relative positions in the finer levels (earlier levels of the transformation), create a binary tree structure. Thus, 34 is the lowest level of the transformation and the root of the points above it.

<table>
<tr><td colspan="16">34</td></tr>
<tr><td colspan="8">22</td><td colspan="8">25</td></tr>
<tr><td colspan="4">15</td><td colspan="4">18</td><td colspan="4">22</td><td colspan="4">10</td></tr>
<tr><td colspan="2">12</td><td colspan="2">8</td><td colspan="2">4</td><td colspan="2">16</td><td colspan="2">12</td><td colspan="2">18</td><td colspan="2">9</td><td colspan="2">7</td></tr>
<tr><td>6</td><td>9</td><td>9</td><td>5</td><td>11</td><td>5</td><td>10</td><td>12</td><td>9</td><td>11</td><td>13</td><td>14</td><td>7</td><td>8</td><td>3</td><td>2</td></tr>
</table>

Figure 12: Four level transformation map.

We will set the thresholding level at 10 and ignore elements of a lesser value. Normally, this would be an absolute value threshold since the coefficients will range above and below zero, but for simplicity we will use only positive values.

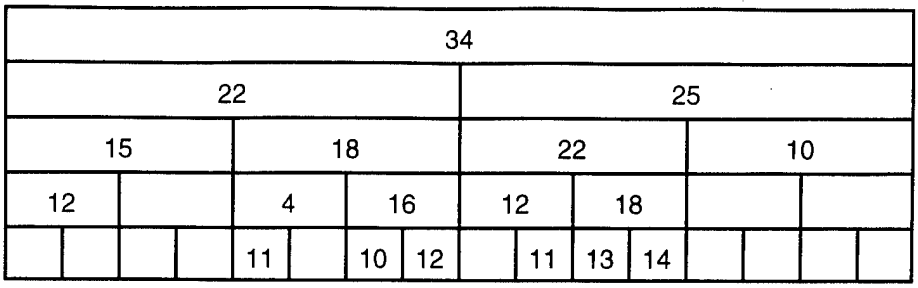

Figure 13: Map after setting threshold at 10.

If we use stop codes to determine the tree structure then for every non-terminal branch node, we must keep one extra node. For every terminal node (except for the last level which we can predetermine) we must keep two extra nodes. The worst case for this model is a full binary tree that does not reach the last level. There will be $n + 1$ stop codes for n nodes kept. This clearly makes the overhead greater than 50% in the worst case. The best case would be a full binary tree where every node is kept and since no stop codes would be necessary the overhead would be zero. Since the purpose of the tree encoding is to reduce the number of nodes stored then this is non-optimal.

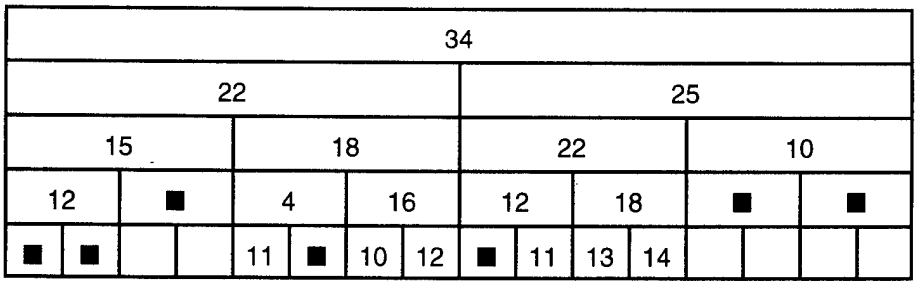

Figure 14: Stop codes for four level transformation map.

The choice then is how many bits to allocate to the decision process. If only one bit is allocated, then it becomes either a "*go/no*" decision on whether to pursue subsequent descendent levels. If the decision is to *go* because one or more of the child nodes are significant, then we must keep both child nodes even if only one is significant. If we allow two bits for the decision we will half the remaining precision left in the byte storage, 64 possibilities instead of 128, but we will improve the overhead of the decision process by reducing the number of subsequent bytes that must be kept, see **Table 3**.

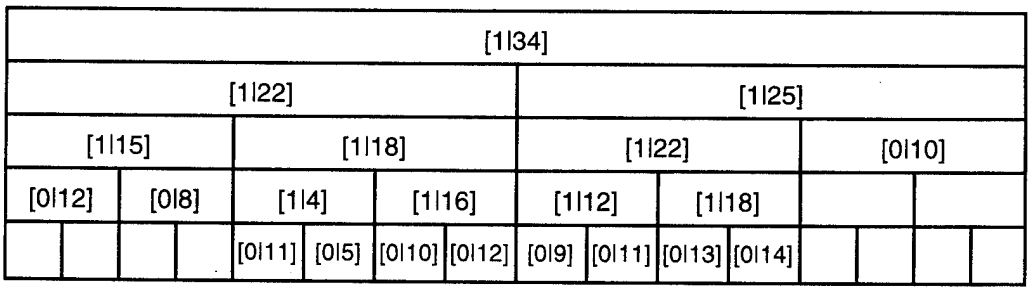

Figure 15: Map with 1-bit binary tree decision prefix.

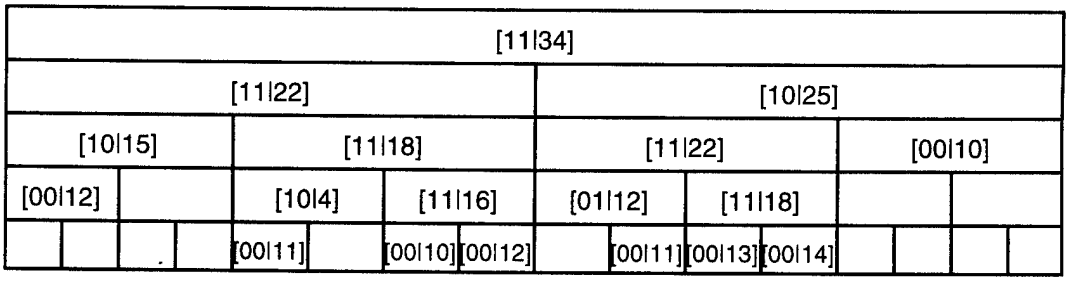

Figure 16: Map with 2-bit binary tree decision prefix.

	1 BIT MODEL			
	CODE	DECISION	EXTRA COST	TOTAL
STOP	0	1	7	8
RIGHT	1	1	23	24
LEFT	1	1	23	24
BOTH	1	1	23	24
SCHEME TOTAL		4	76	80

	2 BIT MODEL			
	CODE	DECISION	EXTRA COST	TOTAL
STOP	00	2	6	8
RIGHT	01	2	14	16
LEFT	10	2	14	16
BOTH	11	2	22	24
SCHEME TOTAL		8	56	64

Table 3: Bit overhead for binary tree decision models.

In the one bit model, if the decision is made to stop at a given point then we have seven remaining bits in the element for the value. If we must continue to the next level because either the left, right, or both subbranches have significant coefficients then both subbranch elements must be kept at an additional cost of two bytes. If the two bit model is used then zero, one, or two additional elements are kept according to the decision. The average overhead for the one bit model is 20 bits while the average overhead for the two bit model is only 16 bits. Accordingly, the scheme is 25% more efficient if the resolution of six bits instead of seven can be used.

Reconstruction is very simple, read the prefix, if there is a 1 in the first bit then continue recursively building the left subtree until a leftmost 0 is encountered. As 0 bits are encountered, go right (the second bit is a 1) or return a level and resume the process. The depth of the reconstruction process is self-limiting and the process can be terminated by exhausting the input.

So [01l..], would represent pruning the left branch and continuing the right branch.

... [11|**34**] [11|**22**] [10|**15**] [00|**12**] [11|**18**][10|**4**] [00|**11**] [11|**16**] [00|**10**]

[01|**12**] [10|**25**] [11|**22**] [00|**12**] [00|**11**] [11|**18**] [00|**13**] [00|**14**] [00|**10**] ...

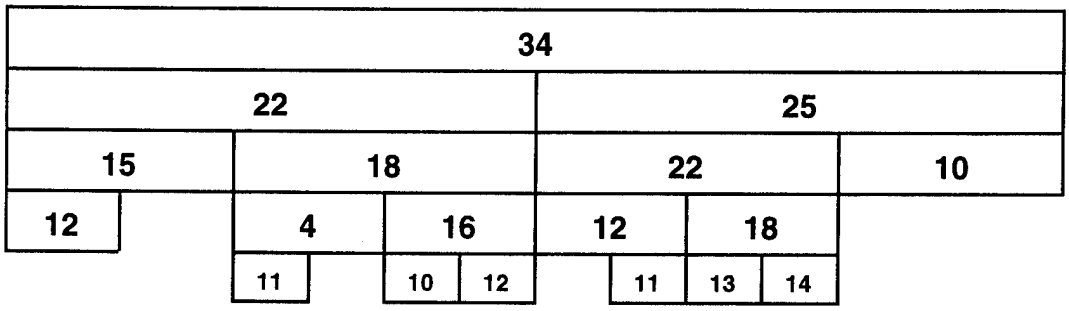

Figure 17: Reconstructed map from coded stream.

Reconstruction is very simple, read the prefix, if there is a 1 in the first bit then continue recursively building the left subtree until a leftmost 0 is encountered. As 0 bits are encountered, go right (the second bit is a 1) or return a level and resume the process. The depth of the reconstruction process is self-limiting and the process can be terminated by exhausting the input.

Chapter 4

Color Images

Up to this point we have been discussing concepts related to 8-bit monochrome greyscale images. The coding of 24-bit color images is more complex issue. At first glance it would appear that we will need three times the efficiency to code the three input channels (R,G,B). In fact, there is a great deal of redundancy in the three color planes and we can reduce the redundancy by performing special transforms on them. A color signal can be transformed into a luminance signal and two color sub-bands as is done in television broadcasting. The bandwidth of the two color sub-bands can be much smaller than the luminance bandwidth to reproduce the spatial detail accurately, perhaps as small as 20%[3]. We can use the reduction in the bandwidth to our advantage, concentrating our best reconstruction efforts on the luminance signal while devoting a much smaller byte budget to the color sub-bands.

Two transforms are in common use today, those of the NTSC (U.S. television standard) and the PAL (European standard) transforms. The NTSC YIQ transform is given by

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(40)}$$

where Y is the monochrome luminance channel and the I and Q channels carry the color opposition signals. The inverse transform is

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.273 & -0.647 \\ 1 & -1.104 & 1.701 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad \textbf{(41)}$$

The PAL transform is similar in that the Y is derived in the same manner, but the color difference signals are directly derived as follows

$$U_d = 0.493(B - Y)$$
$$V_d = 0.877(R - Y) \quad \textbf{(42)}$$

the scale factors are introduced to limit the maximum instantaneous signal amplitude.

An alternative transform corresponding to the CIE uniform chromaticity scale may also be applied, where the V channel corresponds to the Y channel of the YIQ transform.

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} 0.405 & 0.116 & 0.133 \\ 0.299 & 0.587 & 0.114 \\ 0.145 & 0.827 & 0.627 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(43)}$$

With a further conversion of

$$u = \frac{U}{U+V+W} \tag{44}$$

and

$$v = \frac{V}{U+V+W} \tag{45}$$

to give a Y,u,v coordinate system.

Another transform based on a KLT coordinate rotation to generate uncorrelated color components was developed by W. K. Pratt [24] and is given by the orthogonal transform

$$\begin{bmatrix} K_1 \\ K_2 \\ K_3 \end{bmatrix} = \begin{bmatrix} 0.575 & 0.615 & 0.540 \\ 0.608 & 0.120 & -0.785 \\ 0.548 & -0.779 & 0.305 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{46}$$

where the K_1 channel carries the luminance and the $_2K$ and $_3K$ channels carry the color opposition components. Its inverse is given by

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.575 & 0.608 & 0.547 \\ 0.615 & 0.120 & -0.779 \\ 0.539 & -0.785 & 0.305 \end{bmatrix} \begin{bmatrix} K_1 \\ K_2 \\ K_3 \end{bmatrix} \tag{47}$$

From Pratt's work it was found that for an example image the signal energy redistribution is that shown in **Table 4**

System	Component 1 (%)	2 (%)	3 (%)
RGB	33.2	36.2	30.6
YIQ	93.0	5.3	1.7
K_1 K_2 K_3	94.0	5.1	0.9

Table 4: Color signal energy redistribution

As is readily apparent from the table, the dynamic ranges of the color opposition signals of either transform is much smaller than the luminance signal.

An example of the RGB color planes and the K_1, K_2, K_3 component planes from a 24-bit image of fruit are in **Figure 18** and **Figure 19**. In the RGB images it is easy to see how much redundant information is contained in all three subimages. In the K_1, K_2, and K_3 component images the K_1 plane contains the bulk of the image information, while the K_2 and K_3 channels contain minimal information.

It was found that the K_2 and K_3 color opposition channels can be compressed as much as 0.063 bits/pixel and still reconstruct a visually reasonable color map for the image. It is this ability to restore an adequate color map that allows visual degradation in that is unacceptable in a monochrome image to be surpassed by a color reconstruction.

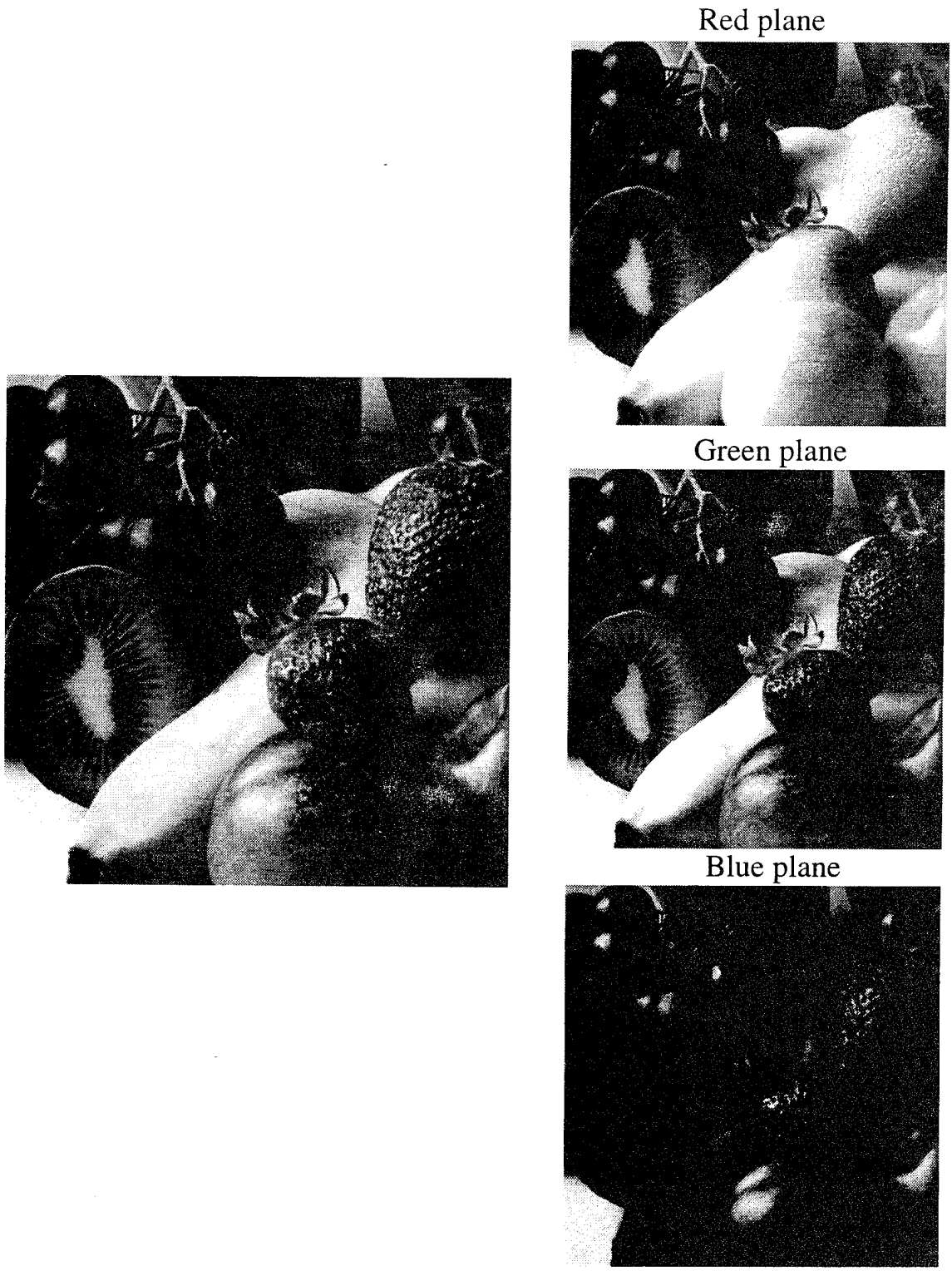

Figure 18: RGB color separation of fruit.

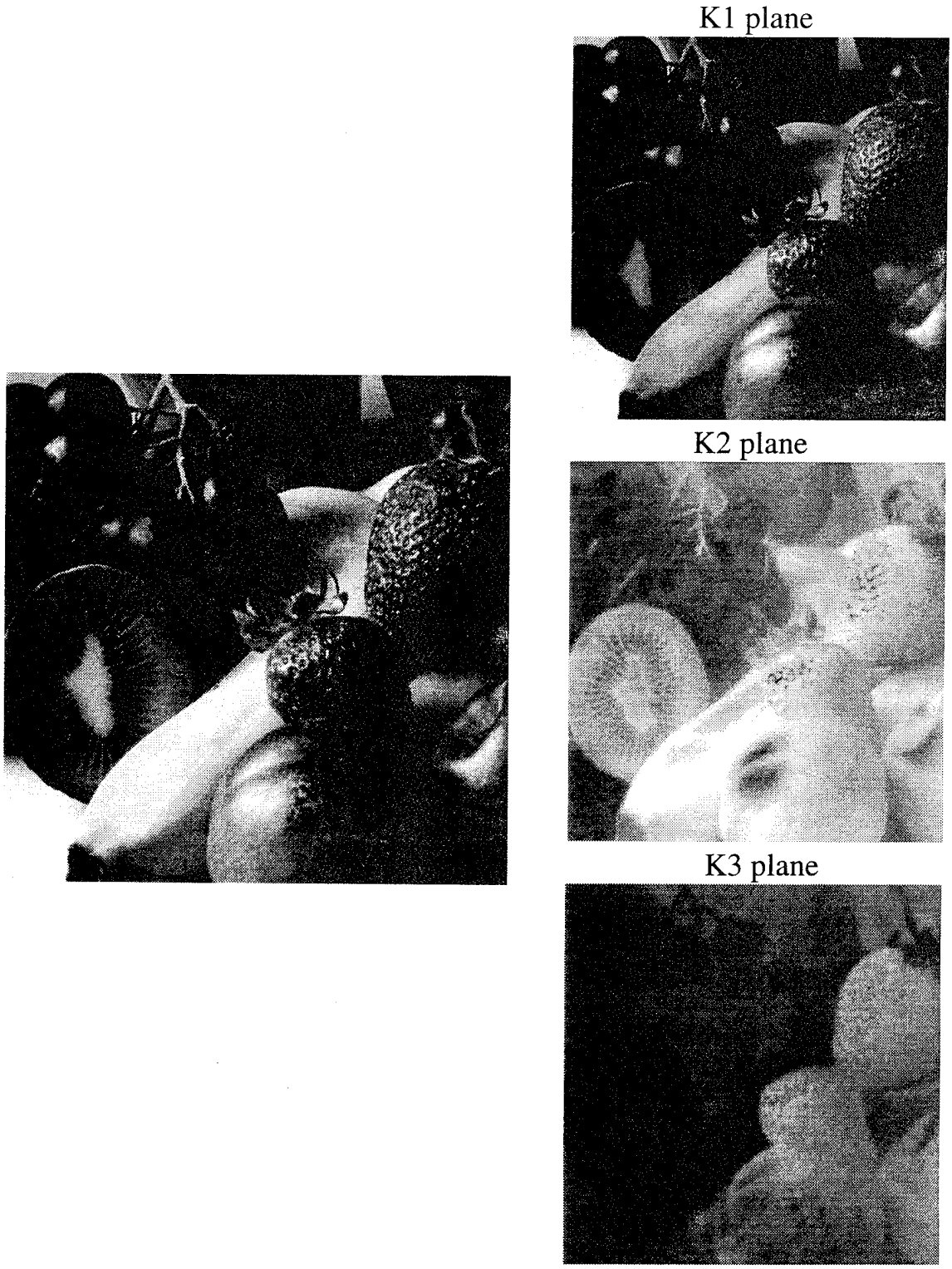

Figure 19: K_1 K_2 K_3 component separation of fruit.

Color transformation can be improved by looking at human visual systems modeling, a logarithmic function/bandpass of the form

$$\begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.607 & 0.174 & 0.201 \\ 0 & 0.066 & 1.117 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(48)}$$

and

$$\begin{bmatrix} G_1 \\ G_2 \\ G_3 \end{bmatrix} = \begin{bmatrix} 21.5 & 0 & 0 \\ -41.0 & 41.0 & 0 \\ -6.27 & 0 & 6.27 \end{bmatrix} \begin{bmatrix} \log T_1 \\ \log T_2 \\ \log T_3 \end{bmatrix} \quad \textbf{(49)}$$

is more effective at converting the color system into one that responds to detectable changes observed by the eye.

The T_1 channel carries the luminance signal and the T_2 and T_3 channels carry the color opposition signals and are invariant to linear scaling of the R, G, and B inputs. The G_2 and G_3 correspond to the red-green and blue-yellow channels. A key feature of this model is that it is very effective in compacting the total R, G, and B color energy selectively into the G_1 channel while leaving the other two channels significantly correlated.

Chapter 5

A Multi-Level Compression Model

Implementation

To test and develop the color image compression model we implemented the components in separate modules much as we have described the development of this paper, see **Figure 20**. It consists of four separate stages: The color transformation stage, dividing the image into three separate sub-planes. The wavelet transform stage, transforming each of the sub-planes with a wavelet transform. Next is the quantization step, converting the transformed planes into a finite representation. The last step is to compress the representation by reducing the number of elements stored. Reconstruction is accomplished by reversing the sequence and using the inverse operators.

We elected to use Pratt's color transform in our implementation because of its simplicity, good energy redistribution, and orthogonality. The original 24-bit image is converted into three planes of floating point values. The transformation is lossless and perfect reconstruction can be obtained from these three planes.

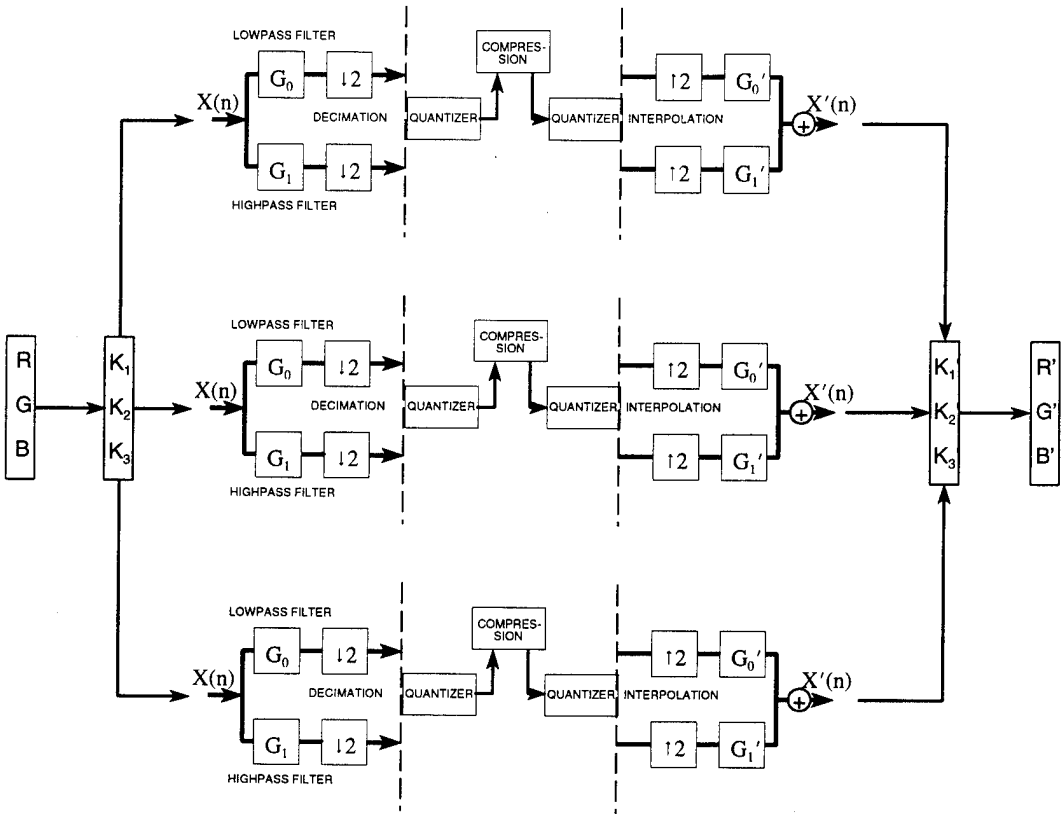

Figure 20: Multi-level color compression model.

More than 90% of the total amount of information is contained in the first or K_1 plane, dictating the luminance and therefore the edges in the image. Hence, it is from this plane that we must retain the greatest information. If a general mapping of the color opposition planes can be obtained, we will be able to approximate the locations and variations of the color. This allows us to sacrifice detail information contained in these planes. Approximately 10%-20% of the data required for reconstruction of the K_1 plane is required for the K_2 and K_3 planes. Care must be taken to ensure that the same level of approximation is used on K_2 and K_3 planes since the balance of color opposition must be maintained to reconstruct a color

scheme that is in the correct range. Unbalancing of these planes will cause undesirable color shifts.

Figure 21: Snaking the image.

A primary consideration for our model was flexibility and the ability to handle various image sizes along with reducing the implemention complexity. Since two dimensional QMF models require images to be in N^2 formats where N is some power of two, we chose to use a separable linear transform method and modified it to handle various image formats. The image was treated as a one dimensional array rather than a two dimensional array. The advantages are that any size image can be processed although it can only be processed to the greatest power of two that will span the image or it must be appended with an array of zeros

to reach the next highest power of two. Usually, the passes of the filter are made across the image boundary, the end of a scan line to the next scan line, this discontinuity of the image creates spikes in the transform domain. These spikes can be misidentified as significant coefficients. To eliminate this edge effect, an option to snake the image was implemented. *Snaking* the image, as is implied in **Figure 21**, takes every other scan line and reverses the order. The image edges are now next to each other creating continuity in the image and the only point that the boundary condition exists is in the wrap around from the end of the image to the first element of the image. This effect can be seen on the image "Lenna", **Figure 22** and **Figure 23**. The wrap around is necessary because as the filter approaches the end of the image it must have sufficient elements to calculate the coefficients. The over run is accomplished by wrapping around to the beginning of the image instead of adding unwanted buffering elements. We implemented a range of filters with which to experiment. These are the Daubechies filters of 4 - 20 coefficients.

Figure 22: Original Lenna image.

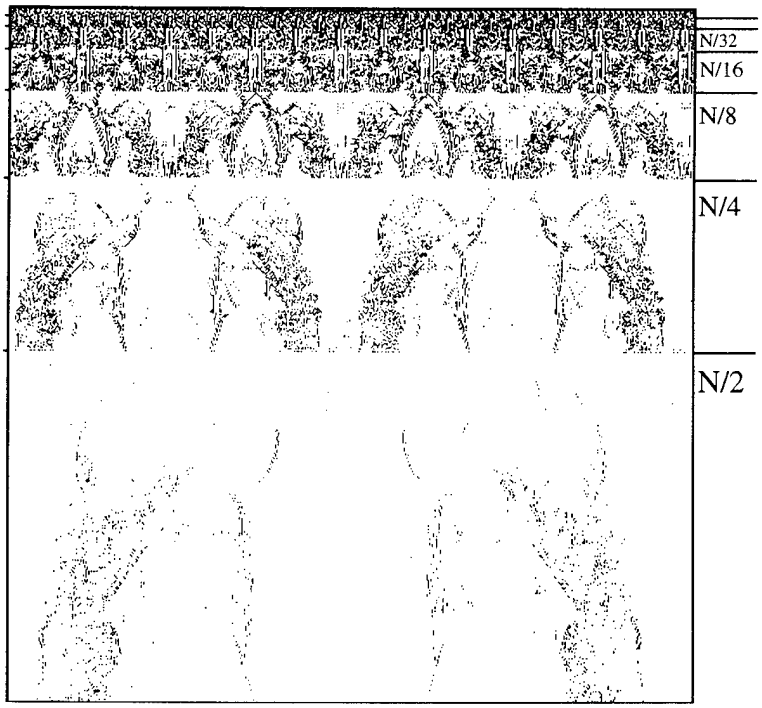

Figure 23: Significant coefficient map of transformed Lenna.

Quantizing is the weakest portion of our model since it is only a linear bin distribution based on the construction of a set of quantizing bins from a fixed number of bins for the given sub-band, see **Table 5**. Since each deeper level of the pyramid contains a greater portion of the energy of the image the number of bins is increased for each level of the transform. This captures about 93% of the values within the bins specified for the level.

This gives reasonable flexibility to the coefficient distribution and reconstruction. The total cost for the extended coding is two and three bytes respectively. Example runs can be seen in **Table 6** and **Table 7**.

```
level 16, sect 1 int quantizing, 100 bins
   mean = 17008.0, var = 725523185.89, stddev = 26935.54
   high = 63614.39, low = -317.83, size =       4, bins = 100
   binsize = 808.07, zeros = 0, longest zero run = 0
   outsiderthresh = 80806.61, outsiders = 0
For level 16, sect 2 quantizing, 100 bins
   mean = 764.5, var = 575880.96, stddev = 758.87
   high = 1744.64, low = -375.55, size =       4, bins = 100
   binsize =  22.77, zeros = 0, longest zero run = 0
   outsiderthresh = 2276.60, outsiders = 0
For level 15, sect 2 quantizing, 93 bins
   mean = 17.1, var = 510768.84, stddev = 714.68
   high = 1032.83, low = -1109.04, size =       8, bins = 93
   binsize =  23.05, zeros = 0, longest zero run = 0
   outsiderthresh = 2144.04, outsiders = 0
For level 14, sect 2 quantizing, 87 bins
   mean =  3.6, var = 93784.47, stddev = 306.24
   high = 545.63, low = -581.80, size =      16, bins = 87
   binsize =  10.56, zeros = 0, longest zero run = 0
   outsiderthresh = 918.73, outsiders = 0
:
For level 6, sect 2 quantizing, 37 bins
   mean =  1.1, var = 32306.60, stddev = 179.74
   high = 662.18, low = -659.70, size =    4096, bins = 37
   binsize =  14.57, zeros = 293, longest zero run = 3
   outsiderthresh = 539.22, outsiders = 19
For level 5, sect 2 quantizing, 31 bins
   mean =  0.4, var =  9105.61, stddev =  95.42
   high = 390.33, low = -372.62, size =    8192, bins = 31
   binsize =  9.23, zeros = 941, longest zero run = 6
   outsiderthresh = 286.27, outsiders = 54
For level 4, sect 2 quantizing, 25 bins
   mean =  0.0, var =  2898.01, stddev =  53.83
   high = 317.94, low = -290.04, size =   16384, bins = 25
   binsize =  6.46, zeros = 3101, longest zero run = 15
   outsiderthresh = 161.50, outsiders = 238
For level 3, sect 2 quantizing, 18 bins
   mean =  0.3, var =  731.22, stddev =  27.04
   high = 181.01, low = -170.25, size =   32768, bins = 18
   binsize =  4.51, zeros = 11975, longest zero run = 32
   outsiderthresh =  81.12, outsiders = 699
For level 2, sect 2 quantizing, 12 bins
   mean =  0.0, var =  169.55, stddev =  13.02
   high = 132.14, low = -170.19, size =   65536, bins = 12
   binsize =  3.26, zeros = 36851, longest zero run = 50
   outsiderthresh =  39.06, outsiders = 1698
For level 1, sect 2 quantizing, 6 bins
   mean =  0.0, var =   26.89, stddev =  5.19
   high =  60.81, low = -64.59, size =  131072, bins = 6
   binsize =  2.59, zeros = 86044, longest zero run = 72
   outsiderthresh =  15.56, outsiders = 2823
```

Table 5: Example run from quantizer.

The purpose of selecting these simple model parameters is to explore the model's binary tree storage capabilities. Exception encoding is employed for values outside this range by encoding a special symbol and adding an additional one or two bytes to the storage. In this manner we could relax the bin partitioning requirements and allow a small percentage of the quantized values to exceed the normal range. The first special code allows one byte storage of values up to ±127. The second special code provides a range of ±32768.

```
levels = 15, direction 1, lookahead = 10, thresh1 = 15, thresh2 = 0
saving header of size 34
setting up tree pointers
maxlevel is 18, levels requested are 15
  loc   18 offset 0 contains 2 floats
  loc   17 offset 2 contains 2 floats
  loc   16 offset 4 contains 4 floats
  loc   15 offset 8 contains 8 floats
  loc   14 offset 16 contains 16 floats
  loc   13 offset 32 contains 32 floats
  loc   12 offset 64 contains 64 floats
  loc   11 offset 128 contains 128 floats
  loc   10 offset 256 contains 256 floats
  loc   9 offset 512 contains 512 floats
  loc   8 offset 1024 contains 1024 floats
  loc   7 offset 2048 contains 2048 floats
  loc   6 offset 4096 contains 4096 floats
  loc   5 offset 8192 contains 8192 floats
  loc   4 offset 16384 contains 16384 floats
  loc   3 offset 32768 contains 32768 floats
  loc   2 offset 65536 contains 65536 floats
  loc   1 offset 131072 contains 131072 floats
8 tokens copied, 8 tokens to process
level 15, left at  0, i=-10, o=54
level 14, left at  0, i=51, o=51
level 13, left at  0, i=-4, o=60
level 12, left at  0, i=-16, o=48
level 11, left at  0, i=-6, o=58
level 10, left at  0, i=-23, o=41
level 9, left at  0, i=0, o=0
level 8, left at  0, i=7, o=7
level 7, right at 0, i=-7, o=-71  -  pruned left, going right
level 6, right at 1, i=0, o=-128  -  pruned left, going right
level 5, stop at  3, i=-15, o=-15   - both pruned, going back
level 8, right at 0
level 7, stop at  1, i=-4, o=-4   - both pruned, going back
level 9, right at 0
level 8, left at  1, i=-3, o=61
level 7, left at  2, i=2, o=2
level 6, left at  4, i=0, o=64 - pruned right, going left
level 5, right at 8, i=-3, o=-67  -  pruned left, going right
level 4, stop at  17, i=16, o=-48   - both pruned, going back
level 6, right at 4 skipped
 :
output file write of 25871
lenna.tki.q2 = 262144 tokens, converted to 25871 char
0 long codes output
output stop = 8608, left = 4041, right = 3913, spec = 385
total code tokens 16947
special token overhead 14.8%
```

Table 6: Binary tree encoding output.

```
levels = 15, direction -1,saving header of size 34
reading in 25984 data tokens
rebuilding tree pointers
maxlevel is 18, levels to restore are 15
  loc   18 offset 0 contains 2 floats
  loc   17 offset 2 contains 2 floats
  loc   16 offset 4 contains 4 floats
  loc   15 offset 8 contains 8 floats
  loc   14 offset 16 contains 16 floats
  loc   13 offset 32 contains 32 floats
  loc   12 offset 64 contains 64 floats
  loc   11 offset 128 contains 128 floats
  loc   10 offset 256 contains 256 floats
  loc   9 offset 512 contains 512 floats
  loc   8 offset 1024 contains 1024 floats
  loc   7 offset 2048 contains 2048 floats
  loc   6 offset 4096 contains 4096 floats
  loc   5 offset 8192 contains 8192 floats
  loc   4 offset 16384 contains 16384 floats
  loc   3 offset 32768 contains 32768 floats
  loc   2 offset 65536 contains 65536 floats
  loc   1 offset 131072 contains 131072 floats
8 tokens copied
level 15, left at  0, i=54, o=-10
level 14, left at  0, i=51, o=51
level 13, left at  0, i=60, o=-4
level 12, left at  0, i=48, o=-16
level 11, left at  0, i=58, o=-6
level 10, left at  0, i=41, o=-23
level 9, left at  0, i=0, o=0
level 8, left at  0, i=7, o=7
level 7, right at 0, i=-71, o=-7  - pruned left, going right
level 6, right at 1, i=-128, o=0  - pruned left, going right
level 5, stop at  3, i=-15, o=-15  - both pruned, going back
level 8, right at 0
level 7, stop at  1, i=-4, o=-4  - both pruned, going back
level 9, right at 0
level 8, left at  1, i=61, o=-3
level 7, left at  2, i=2, o=2
level 6, left at  4, i=64, o=0 - prune right, going left
level 5, right at 8, i=-67, o=-3  - pruned left, going right
level 4, stop at  17, i=-48, o=16  - both pruned, going back
level 6, right at 4 skipped
:
level 15 ,size 8, index 8 completed
lenna.tki.q2.zt2 = 25984 tokens, converted to 25486 floats
output file write of 262144
```

Table 7: Binary tree reconstruction output.

A visual example of the binary tracing of the coefficients is in **Figure 24**. When looking at this example keep in mind that each level is actually one half the size of the previous level. In the Lenna example, **Figure 25**, the image width spreads these blocks across the original width of the image. Tracings are used to give an idea of the relationship of the blocks to each other.

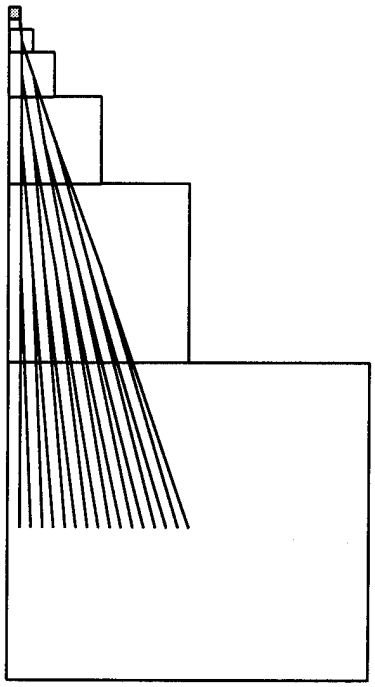

Figure 24: Binary relationship between blocks.

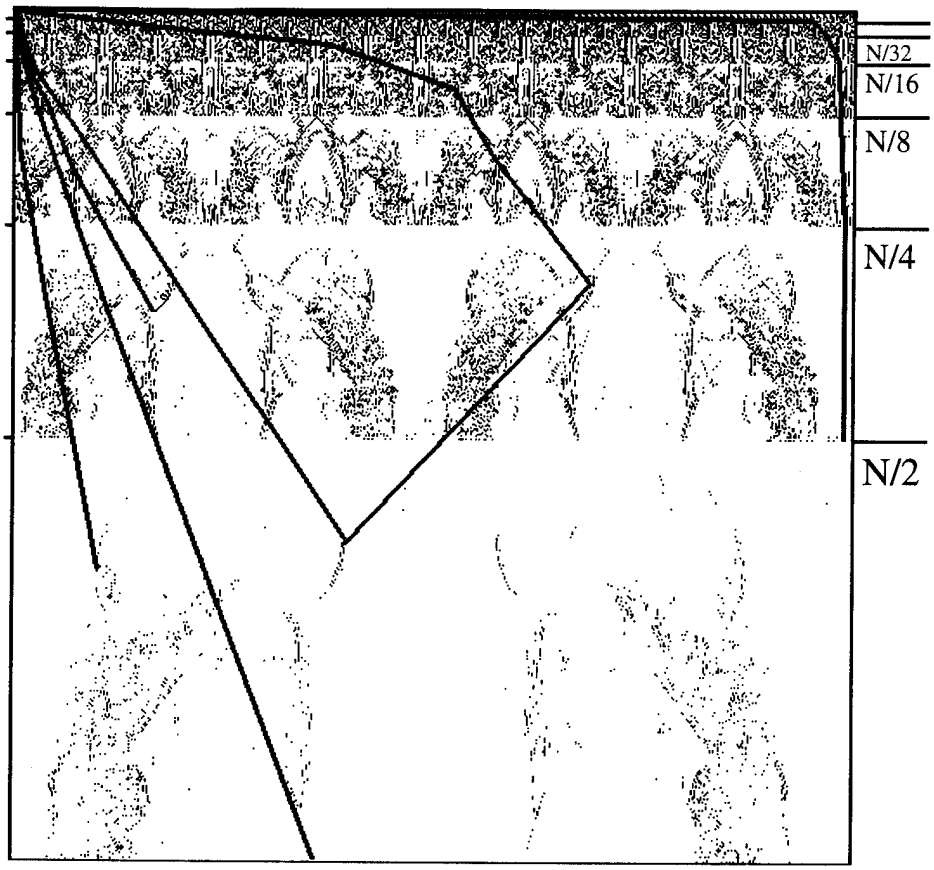

Figure 25: Example coefficient binary tree relationship tracings.

An example of the restored image of lenna after 8.8:1 compression is in **Figure 26**. At this level of compression the losses are very minor. Note the loss in the area around the top of the hat along with some edge degradation around the frame of the mirror and the feathers of the hat as compared to the image in **Figure 22**.

Figure 26: Restored image of Lenna after 8.8:1 compression.

We evaluated the performance of the color transforms and the K_1, K_2, K_3 transform provided the best performance among it and the CIE, YIQ, and PAL transforms. Because of the crudeness of the binary tree model it was necessary to switch out the compression engine to effectively evaluate the color transformations. By changing the compression engine to a two-dimensional run-length model compression ratios were significantly enhanced. We found the K_1, K_2, K_3 model to be the least sensitive to color component degradation as long as the K_2 and K_3 channels are compressed with the same parameters. The K_3 channel generally required only half the number of coefficients as K_2channel. Since the K_1 channel carries more than 90% of the information, the other two channels can be compressed to less than half of the K_1 channel.

Performance

The binary tree storage proved to be a viable concept, compression ratios of 10:1 for 8-bit images were achieved even though the model only used a horizontal pass for the pyramid construction. With the addition of horizontal and vertical passes, energy movement should be significantly greater and the coefficient reduction in the highband pass should be closer to zero. The most difficult process is the discrimination of significant coefficients between levels of the pyramid. Since the energy is packed into fewer coefficients at each level, the coefficients in the lower levels of the pyramid contribute more to the overall energy of the image than those of the higher levels. Care must be used when eliminating coefficients at the lower levels because the removal can cause significant dropouts in the affected portions of the image. Conversely, the coefficients at the higher levels establish the details of the image and must be carefully retained to prevent washout of the image.

Although the quantizer used is very crude, it was found that by first determining significance across the levels of the pyramid and then increasing the number of bins used by the quantizer for each level of the pyramid processed, permitted better discrimination and better approximation of the true value. Using soft thresholding permits the storage capacity of the token to be maximized.

Use of the binary tree storage technique from our model provided only modest color compression on the order of 15:1 to 20:1. More importantly though, when the compression engine was replaced with a model that used a more sophisticated two dimensional transform with run length encoding, it was found that the color separation transforms could be used to achieve average compression ratios of 60:1 with only minor visual defects. It was possible to reach ratios of 130:1 with only modest degradation in some images before the structural overhead of the storage method limited the performance.

A suite of ten varied color images, all 512 x 512 and 24-bit color, were used for testing. The worst compression was 40:1, albiet with excellent reconstruction. The image was of a tiger and the fur of the tiger acts like random noise. This created a lot of detail coefficients the transformed image which reduced the compressibility. The best compression was 158:1 with some noticeable, but not unreasonable image degradation. This image was of a sunset in clouds. Detail in the clouds was lost at this high of a compression ratio.

We also tested our method on video images. Using captured video frames, compression ratios of 70:1 to 80:1 were easily achieved. The video images suffer from some visual defects that are not present in the still images. These defects are a result of motion blur and of the poorer resolution of video signals from video tape and tend to appear in the edge detail of the image. Since these losses complements those normally generated by the wavelet transform we were able to get excellent results.

These color compression ratios easily exceed the 20:1 compression provided by the JPEG standard. With further research on the optimal color and wavelet transforms, and standardization of the underlying encoding structure, wavelet transformation provides a viable and attractive alternative to supercede JPEG.

Chapter 6

Conclusion

We now conclude this thesis by summarizing the major points and providing some directions for future research.

Summary

In the thesis we have presented a survey of the major forms of spatial compression techniques along with exploring the transform domain techniques. We have seen that there is a variety of wavelet transform techniques and discussed the basic operating principles.

We presented a flexible wavelet transform model that permits compression techniques to be explored on a variety of image sizes, either in greyscale or color. Although the binary tree model did not implement the most effective of the transform methods discussed, it does allow the compression techniques to be explored on a range of images.

It was found that color image compression can be effectively implemented by using wavelet transform techniques and can surpass the techniques that are in common use today in the JPEG standard. The keys to successful color image compression are transforming the image into a representation that minimizes the channel bandwidth required to reconstruct the RGB color planes of the image. By using models that represent the luminance and color opposition channels effectively, lossy compression can be tolerated at levels beyond lossy monochrome compression because the visual degradation is masked by color information.

It was also determined that another key to successful compression is the selection of effective quantization and storage techniques.

Further Research

Although the binary tree storage representation implemented was not as effective as is expected, it is known that this is in part due to the inadequate quantization techniques. If better quantization techniques are brought to bear on the problem and heuristic thresholding techniques are used, it is believed that binary tree representation of the transformed image can surpass standard run-length encoding of the image. Improvements in the quantization of the data can be made by minimizing the mean square error or embedding the quantizer after determining the significance of a coefficient.

Another viable solution that was not explored with perhaps better discrimination would be to first soft threshold the transformed data, build the binary tree, and then quantize the remaining data based on the only the retained coeffients. This would eliminate from consideration both isolated coefficients and coefficients below the threshold when constructing the quantization bins.

Human visual models that minimize the detectable degradation of an image must be further explored. Due to time limitations, the logarithmic transform was not tested against the other color transfoms to determine if it creates a more visually appealing image at higher compression ratios. If appropriate transforms are developed that allow quantization errors to be greatest within ranges that are less detectable by the human eye then compression ratios can reach even greater levels.

Further research must be done on the underlying the underlying structure of the storage method. As we reach greater levels of compression, the actual structure overhead then becomes the limiting factor. Some of the structure can be hard coded in the compression model rather than being passed along with the data. Reducing the precision of the control information passed along in the model from 32-bit floating point values to 16-bit or 8-bit values can also be used to reduce overhead.

Lastly, it must be determined how much loss in the image can be tolerated with real time video. The movement between frames and the 30 frames/second refreshing of a video image limits the eye's ability to detect flaws, it may be possible to sustain a compression

ratio of over 100:1 in video compression without significantly affecting the visual quality. Also, methods of intraframe compression can be explored by only saving and compressing the differences between the frames instead of the entire next frame.

Implementation of the wavelet transform in hardware should perform better and faster than today's JPEG because of the reduced complexity of the wavelet transform over the DCT. By using a pyramid wavelet transform it will take $O(N)$ versus the $O(N \log N)$ of the DCT.

Appendix I

Computer Program Listings

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: RGBKKK.C

PURPOSE: Splits a RGB image into three planes, a luminance
plane and two color opposition planes

Input file is a RGB (24-bit) image file
3 output files are created each is a float file


FEATURES: can handle windows BMP format files also
-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>


/********************MAIN*************************************/
int main(int argc,char *argv[])
{
 FILE *in, *R, *G, *B, *Y, *I, *Q, *YT, *IT, *QT;
 int i,j,bmp=0;
 char filename[12],othername[12]={0,0,0,0,0,0,0,0,0,0,0,0},
   name[8]={0,0,0,0,0,0,0,0},
   ext[4]={0,0,0,0},
   headerbuf[100];
 unsigned char RGB[3],KKK2[3];
 float KKK[3];
 int width,height;
 unsigned long count=0;
 long size;
 int textout=0;
```

```
if (argc<2) printf("usage: pgm filename [t] {for text output}\n");
if (argc>2&&argv[2][0]=='t') textout=1;
strcpy(name,argv[1]);

if ((in = fopen(argv[1],"rb")) == NULL)
{
 fprintf(stderr,"Input %s not found.\n",filename);
 exit(1);
}
for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,argv[1]);
strncat(othername,".res",4);

if (strcmp(ext,"bmp")==0)  /* assume BMP file */
{
 fread(headerbuf,1,57,in);

 if (headerbuf[0]!='B'|| headerbuf[1]!='M')
 {
  printf("File not windows bitmap\n");
  exit(0);
 }

 if (headerbuf[28]!=24)
 {
  printf("Image is not 24 bit color windows bitmap\n");
  exit(0);
 }

 bmp =1;
 width = (((int) headerbuf[19])<<8) + (int) headerbuf[18];
 height = (((int) headerbuf[23])<<8) + (int) headerbuf[22];
}
else /* assume RGB file */
{
 size=filesize(in);
 height=width=(int)sqrt((double)size/3);
 if((long)height*width*3-size!=0)
 {
  printf("filesize error\n");
  exit(1);
 }
}

printf("height=%i, width=%i, size=%lu\n",height,width,(unsigned long)height*width);

for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".k1f",4);

if ((Y = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
```

```
}

for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".k2f",4);

if ((I = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
}
for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".k3f",4);

if ((Q = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
}

for(i=0;i<height*width;i++)
{
 fread(RGB,1,3,in);

 if (bmp)
 {
  KKK[0]=(0.575*(float)RGB[2]+0.615*(float)RGB[1]+0.540*(float)RGB[0]);
  KKK[1]=(0.608*(float)RGB[2]+0.120*(float)RGB[1]-0.785*(float)RGB[0]);
  KKK[2]=(0.548*(float)RGB[2]-0.779*(float)RGB[1]+0.305*(float)RGB[0]);
 }
 else
 {
  KKK[0]=(0.575*(float)RGB[0]+0.615*(float)RGB[1]+0.540*(float)RGB[2]);
  KKK[1]=(0.608*(float)RGB[0]+0.120*(float)RGB[1]-0.785*(float)RGB[2]);
  KKK[2]=(0.548*(float)RGB[0]-0.779*(float)RGB[1]+0.305*(float)RGB[2]);
 }

 /* this is for char based files, scaling and tranlation functions to fit back into char files
 KKK2[0]=(unsigned char)(KKK[0]/1.73);
 KKK2[1]=(unsigned char)(KKK[1]/1.513+132.3);
 KKK2[2]=(unsigned char)(KKK[2]/1.632+121.7);
 fwrite(&KKK2[0],1,1,Y);
 fwrite(&KKK2[1],1,1,I);
 fwrite(&KKK2[2],1,1,Q);
 */

 fwrite(&KKK[0],sizeof(float),1,Y);
 fwrite(&KKK[1],sizeof(float),1,I);
 fwrite(&KKK[2],sizeof(float),1,Q);

 if (textout)
 {
  fprintf(YT,"%6.2f ",KKK[0]);
```

```
      fprintf(IT,"%6.2f ",KKK[1]);
      fprintf(QT,"%6.2f ",KKK[2]);
    }
    if (count++%width==0)
    {
      printf("%lu\r",count);

    }
  }
  printf("\n");
  fclose(in);
  fclose(Y);
  fclose(I);
  fclose(Q);
  return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: KKKRGB.C

PURPOSE: Reconstructs an RGB file from 3 float files, recovers
RGBKKK's transform

-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>


/********************MAIN************************************/
int main(int argc,char *argv[])
{
 FILE *out, *K1, *K2, *K3;
 int i,j;
 char filename[20],othername[12]={0,0,0,0,0,0,0,0,0,0,0,0},
   name[8]={0,0,0,0,0,0,0,0},
   ext[4]={0,0,0,0},
   headerbuf[100];
 unsigned char KKK2[3];
 float RGB[3],KKK[3];
 int width,height;
 unsigned long count=0;


 if ((out = fopen(argv[1],"wb")) == NULL)
 {
  fprintf(stderr,"output %s not created.\n",filename);
  exit(1);
 }

 for (i=0;i<=14;i++) othername[i]=0;
 strcpy(othername,argv[1]);
 strncat(othername,".k1f",4);

 if ((K1 = fopen(othername,"rb")) == NULL)
 {
  fprintf(stderr,"input file %s not found.\n",othername);
  exit(1);
 }

 for (i=0;i<=14;i++) othername[i]=0;
 strcpy(othername,argv[1]);
 strncat(othername,".k2f",4);

 if ((K2 = fopen(othername,"rb")) == NULL)
 {
```

```
  fprintf(stderr,"input file %s not found.\n",othername);
  exit(1);
 }
 for (i=0;i<=14;i++) othername[i]=0;
 strcpy(othername,argv[1]);
 strncat(othername,".k3f",4);

 if ((K3 = fopen(othername,"rb")) == NULL)
 {
  fprintf(stderr,"input file %s not found.\n",othername);
  exit(1);
 }

 while (!feof(K1))
 {
  fread(&RGB[0],sizeof(float),1,K1);
  fread(&RGB[1],sizeof(float),1,K2);
  fread(&RGB[2],sizeof(float),1,K3);
  if(feof(K1)) break;


  KKK[0]=0.575*RGB[0]+0.608*RGB[1]+0.547*RGB[2];
  KKK[1]=0.615*RGB[0]+0.120*RGB[1]-0.779*RGB[2];
  KKK[2]=0.539*RGB[0]-0.785*RGB[1]+0.305*RGB[2];

  KKK2[0]=(unsigned char)(KKK[0]+.5);
  KKK2[1]=(unsigned char)(KKK[1]+.5);
  KKK2[2]=(unsigned char)(KKK[2]+.5);

  fwrite(KKK2,1,3,out);

  if (count++%1000==0)
  {
   printf("%lu\r",count);
  }
 }

 printf("\n");
 fclose(out);
 fclose(K1);
 fclose(K2);
 fclose(K3);
 return 0;
}
```

```
/*-------------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------------

FILENAME: ZT2.C

PURPOSE:  Builds/recovers a zero subtree from a wavelet
transform file

Input file is a binary float file
output file is a char file

Expects a data header of 1 + 2*levels processed (floats)
  Containing levels processed and mean and standard deviation of each
  level processed.

FEATURES: intelligent selection of data nodes to keep uses
a 2-bit binary prefix to encode directional decision data

CONSTRAINTS:  no file size detection routines

-------------------------------------------------------------
-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define DEBUG  0

/*************************************************************
GLOBALS
*************************************************************/
FILE *infile, *outfile;
int outcount = 0;
int count126 = 0;
int count127 = 0;
int countm127= 0;
int countspec= 0;
int countlong= 0;
int threshold1 = 0;
float threshold2 = 0;
int lookahead = 2;
int hsize = -1;
int level = 0;
int direction = 1;
int info = 1;
int dbgcount=100;
int dbg=0;
int DEBUG2 = 0;

/*************************************************************/
void setoptions(int argc,char *argv[])
{
```

```
  int i;

  if(argc < 4)
  {
    printf("Zero Tree - zero tree for a wavelet transform file  \n");
    printf("This program is for float format files \n");
    printf("Usage: pgm filename LEVELS DIR [options]\n");
    printf("DIR 1 forward -1 backward       \n");
    printf("LEVELS levels to process   \n");
    printf("options:                \n");
    printf("-T# primary   threshold { 0 default }    \n");
    printf("-t# secondary threshold { 0 default }  \n");
    printf("-a# lookahead levels    { 2 default }\n");
    printf("-h# size of header { checks file by default } \n");
    exit(0);
  }

  for (i=4;i<argc;i++)
  {
    switch (argv[i][1])
    {
      case 'a':lookahead = atoi(&argv[i][2]);
               break;
      case 'T':threshold1 = atoi(&argv[i][2]);
               break;
      case 't':threshold2 = atof(&argv[i][2]);
               break;
      case 'h':hsize = atoi(&argv[i][2]);
               info = 0;
               break;
      case 'd':DEBUG2 = atoi(&argv[i][2]);
               break;
    }
  }

  level = atoi(argv[2]);
  direction = atoi(argv[3]);
  if (!abs(direction)==1) direction= 1;
  if (direction!=1) printf("levels = %i, direction %i\n",level,direction);
  else printf("levels = %i, direction %i, lookahead = %i, thresh1 = %i,"
           " thresh2 = %i\n",level,direction,lookahead,threshold1,
           threshold2);
}

/*********************************************************/
void * getvect(int n,int size)
{
  int i;
  void *vectptr;

  vectptr = (void *) calloc(n,size);
  if(vectptr == NULL) {
    printf("Error: dynamic memory fail\n");
    exit(0);
```

```
 }
 return(vectptr);
}

/*********************************************************/
/* file size determinator */
int  filesize(FILE *stream)
{
 int  curpos, length;
 curpos = ftell(stream);
 fseek(stream, 0L, SEEK_END);
 length = ftell(stream);
 fseek(stream, curpos, SEEK_SET);
 return length;
}


/*************************************************************************/
signed char *encode(signed char *treeptr,float *input)
{
 int lowbyte=0xff;

 if(abs((int)*input)<128)                /* -128 is reserved code*/
 {
  *treeptr++ = (signed char)*input;
  outcount++;
 }
 else if (abs((int)*input)<=32895)
 {
  *treeptr++=-128;
  *treeptr++= (signed char)((int)*input>>8);
  *treeptr++= (unsigned char)((int)*input&lowbyte);
  outcount+=3;
  countlong++;
  if(DEBUG) printf("decomp %i into %i + %i\n",(int)*input,
            (int)*(treeptr-2),(int)*(treeptr-1));
 }
 else
 {
  printf("ABORTED: range error exceeded on abs calc = %i\n",(int)*input);
  fwrite(&treeptr[-outcount],sizeof(signed char),outcount,outfile);
  printf("%i token saved in outfile\n",outcount);
  exit(0);
 }
 return treeptr;
}


/*************************************************************************/
int threshcheck(float *loc[],int level,int depth,int retval)
{
 int i, chkval,breadth;
 breadth = 1<<(depth-1);
```

```
 chkval = (threshold1-(int)(threshold2*depth>0)) ?
       threshold1+(int)(threshold2*depth):0;

 if (!(retval&0x02))  /* don't repeat check if already found one */
 {
  for (i=0;i<breadth;i++)  /*left side*/
  {
   if(abs(loc[level][i])>=chkval)
   {
    retvall=0x02;
    if (DEBUG   ) printf("left trigger at lev %i, off %i, ckval %i,"
                  " val %i\n",level,i,chkval,(int)loc[level][i]);
   }
  }
 }
 if (!(retval&0x01))  /* don't repeat check if already found one */
 {
  for (i=breadth;i<breadth*2;i++) /* right side*/
  {
   if(abs(loc[level][i])>=chkval)
   {
    retvall=0x01;
    if (DEBUG   ) printf("right trigger at lev %i, off %i, ckval %i,"
                  " val %i\n",level,i,chkval,(int)loc[level][i]);
   }
  }
 }
 if (retval==3) return retval; /* both sides signif, stop */
 else if (depth<lookahead&&level>1)
    return threshcheck(loc,level-1,depth+1,retval);
 else return retval;
}


/*************************************************************************/
signed char *treetraverse(float *vector,signed char *treeptr, float *loc[],
                  float *tree[],int level)
{
 int i;
 int nextlevel;
 int initval =0;
 int initdepth =1;
 signed char temp;

 /*encode left node and subtree if not zero*/

 if (/*abs(*loc[level])<=threshold1&&*/level>1)
 {                       /* need to see if anything sig is below*/
  switch(threshcheck(loc,level-1,initdepth,initval))
  {
   case 0: /* both subtrees insignificant */
      temp = (signed char) *loc[level];            /* get input */

      if (abs(*loc[level])>31)    /* if beyond range, special code */
```

```
    {
      *treeptr++ = 0xe0;  /* dir bits and output range exception */
      temp = treeptr[-1];              /* looks like 11100000 */
      *treeptr++ = (signed char) *loc[level];
      countspec++;
      outcount+=2;
    }
    else  /* fits in range */
    {                              /* 100000000 -> 00100000 */
     if (temp&0x80) temp |= 0x20;          /* move the sign bit */
     temp |= 0xc0;           /* mask directional bits 11000000 */
     *treeptr++ = temp;                    /* save encoded char */
     outcount++;
    }
    count126++;

    if (DEBUG2-->0)
    {
     printf("level %i, stop at  %i, i=%i, o=%i   - both pruned, "
          "going back\n", level,(int)(loc[level]-tree[level]),
          (int)(signed char)*loc[level],(int)temp);
    }

    nextlevel = level;       /*update subtree pointers 2 ea level*/
    for(i=nextlevel;i>0;i--)
    {
     loc[i] += 1<<(nextlevel-i);
    }
    return treeptr;

  case 1: /* left subtree insignificant, process right*/
    temp = (signed char) *loc[level];              /* get input */

    if (abs(*loc[level])>31)    /* if beyond range, special code */
    {
      *treeptr++ = 0xa0;  /* dir bits and output range exception */
      temp = treeptr[-1];              /* looks like 10100000 */
      *treeptr++ = (signed char) *loc[level];
      countspec++;
      outcount+=2;
    }
    else  /* fits in range */
    {                              /* 100000000 -> 00100000 */
     if (temp&0x80) temp |= 0x20;          /* move the sign bit */
     temp &= 0x3f;           /* clear directional bits 00111111 */
     temp |= 0x80;           /* mask directional bits 10000000 */
     *treeptr++ = temp;                    /* save encoded char */
     outcount++;
    }
    count127++;

    if (DEBUG2-->0)
    {
     printf("level %i, right at %i, i=%i, o=%i  -  pruned left, "
```

```
        "going right \n", level,(int)(loc[level]-tree[level]),
         (int)(signed char)*loc[level],(int)temp);
  }

  /*update subtree pointers, before doing right */
  loc[level]++;          /* consume input */
  nextlevel = level-1;
  for(i=nextlevel;i>0;i--)
  {
   loc[i] += 1<<(nextlevel-i);
  }
  treeptr = treetraverse(vector,treeptr,loc,tree,level-1);
  return treeptr;

case 2: /* right subtree insignificant, process left*/
  temp = (signed char) *loc[level];          /* get input */

  if (abs(*loc[level])>31)    /* if beyond range, special code */
  {
    *treeptr++ = 0x60;  /* dir bits and output range exception */
    temp = treeptr[-1];          /* looks like 01100000 */
    *treeptr++ = (signed char) *loc[level];
    countspec++;
    outcount+=2;
  }
  else  /* fits in range */
  {                          /* 100000000 -> 00100000 */
   if (temp&0x80) temp |= 0x20;        /* move the sign bit */
   temp &= 0x3f;          /* clear directional bits 00111111 */
   temp |= 0x40;          /* mask directional bits 01000000 */
   *treeptr++ = temp;                 /* save encoded char */
   outcount++;
  }
  if (DEBUG2-->0)
  {
   printf("level %i, left at  %i, i=%i, o=%i - pruned right, "
        "going left  \n", level,(int)(loc[level]-tree[level]),
       (int)(signed char)*loc[level],(int)temp);
  }
  countm127++;
  treeptr = treetraverse(vector,treeptr,loc,tree,level-1);

  if (DEBUG2-->0) printf("level %i, right at %i skipped\n",level,
            (int)(loc[level]-tree[level]));
  /*update subtree pointers, after left done */
  nextlevel = level-1;
  for(i=nextlevel;i>0;i--)
  {
   loc[i] += 1<<(nextlevel-i);
  }
  loc[level]++;
  return treeptr;

/* case 3: both sides are significant fall through and process */
```

```
  }
 }

 /* or else process normally */

 temp = (signed char) *loc[level];
 if (DEBUG>0) printf("in temp = %i,",(int)temp);
 if (abs(*loc[level])>31)          /* if beyond range, special code */
 {
  *treeptr++ = 0x20;        /* dir bits and output range exception */
  temp = treeptr[-1];                    /* looks like 00100000 */
  if (DEBUG>0) printf(" out code = %i, ",(int)temp);
  *treeptr++ = (signed char) *loc[level];
  countspec++;
  outcount+=2;
  if (DEBUG>0) printf("out temp = %i\n",(int)temp);
 }
 else  /* fits in range */
 {                                  /* 100000000 -> 00100000 */
  if (temp&0x80) temp |= 0x20;               /* move the sign bit */
  temp &= 0x3f;               /* clear directional bits 00111111 */
  *treeptr++ = temp;                      /* save encoded char */
  outcount++;
  if (DEBUG>0) printf(" out temp = %i FITS\n",(int)temp);
 }

 if(level>1)
 {

  if (DEBUG2-->0) printf("level %i, left at  %i, i=%i, o=%i\n",
       level,loc[level]-tree[level],(int)((signed char)*loc[level]),
       (int)((signed char)*(treeptr-1)));

  treeptr = treetraverse(vector,treeptr,loc,tree,level-1);

  /*encode right subtree if level>1*/

  if (DEBUG2-->0) printf("level %i, right at %i\n",level,
                loc[level]-tree[level]);

  treeptr = treetraverse(vector,treeptr,loc,tree,level-1);
 }
 else
 {
  if (DEBUG2-->0) printf("level %i, done  at %i, i=%i, o=%i\n",
       level,loc[level]-tree[level],(int)((signed char)*loc[level]),
       (int)((signed char)*(treeptr-1)));
 }
 loc[level]++; /* update location */
 return treeptr;
}


/*************************************************************************/
```

```
void zerotree(float *vector,signed char *treevector,int size,int levels)
{
 int i, maxlevel;
 float **loc;
 float **tree;
 signed char *treeptr;

 printf("setting up tree pointers\n");

 i=0;
 while(1<<i<size) maxlevel=++i;
 loc = getvect(maxlevel+1,sizeof(int));
 tree = getvect(maxlevel+1,sizeof(int));

 printf("maxlevel is %i, levels requested are %i\n",maxlevel,levels);

 if(levels>=maxlevel)
 {
   printf("too many levels requested\n");
   exit(0);
 }

 tree[maxlevel] = vector; /* should add test for 1> 2**levels */
 loc[maxlevel] = vector;

 for (i=0;i<maxlevel;i++) /* set up pointers 0 = end, maxlevel = start*/
 {
  tree[i] = &vector[size>>i] ;
  loc[i] = tree[i];
 }

 if(DEBUG)printf("vector address = %i\n",(int)vector);

 for (i=maxlevel;i>0;i--)
 {
  printf("  loc   %i offset %i contains %i floats",
        i,loc[i]-loc[maxlevel],loc[i-1]-loc[i]);
  if(DEBUG) printf(", starts at addr %i",(int)loc[i]);
  printf("\n");
 }

 treeptr = treevector;

 i=0;
 while (&vector[i]< tree[levels])
 {
  treeptr = encode(treeptr,&vector[i]);
  if(DEBUG) printf("vect = %6.2f gives c =%i \n",
                vector[i],(int)treeptr[-1]);
  i++;
 }

 printf("%i tokens copied\n",i);
 printf("%i tokens to process\n",tree[levels-1]-loc[levels]);
```

```
 while (loc[levels]<tree[levels-1]) /* begin recursive tree search call */
 {
  treeptr = treetraverse(vector,treeptr,loc,tree,levels);
 }

}

/*************************************************************************/
signed char *decode(signed char *treeptr,float *outputloc)
{
 int temp;

 if (*treeptr==-128)   /* rebuild integer value */
 {
  treeptr++;
  temp = (int)((signed char)*treeptr++);
  temp <<=8;
  temp += (int)((unsigned char)*treeptr++);
  *outputloc = (float)temp;
  outcount++;
  if (DEBUG) printf("rebuild %i from %i + %i\n",
       (int)*outputloc,(int)*(treeptr-2),(int)*(treeptr-1));
 }
 else
 {
  *outputloc=(float)*treeptr++;
  outcount++;
 }
 return treeptr;
}

/*************************************************************************/
signed char *buildtree(float *tree[], signed char *treevector,int size,
           signed char *treeptr, int level,int offset)
{
 signed char *treelast, ctemp, direction;
 int temp;

 if (offset>=tree[level-1]-tree[level]) /* if input exhasted go back */
 {
  printf("level %i ,size %i, index %i completed\n",
       level,tree[level-1]-tree[level],offset);
  return treeptr;
 }
                        /* get direction and then decode char */
 direction = (*treeptr>>6)&0x03;      /* shift 6, mask with 00000011 */
 ctemp = *(treeptr++)&0x3f;          /* mask out direction 00111111 */
 outcount++;                            /* consume token */
 if(DEBUG>0) printf("ctempin = %i, ",(int)ctemp);
 if (ctemp==0x20) /* then special code exception, rebuild number */
 {
  tree[level][offset] = *treeptr++;       /* output whole next char */
                                        /* consume token */
 }
```

```
else  /* convert this code back */
{
 if(ctemp&0x20) ctemp = ctempl0xe0;       /* if -, expand sign bit */
 tree[level][offset] = (float) ctemp;     /* output value of token */
}

if (DEBUG>0) printf("*treeptr= %i, direction = %i, ctemp = %i\n",
              (int)*(treeptr-1), (int) direction, (int)ctemp);

switch (direction)            /* now decide which direction to go */
{
 case 3:                     /* 11XXXXXX -> 00000011 stop code */
   if (DEBUG2-->0)
   {
    printf("level %i, stop at  %i, i=%i, o=%i  - both pruned,"
         " going back  \n",
         level,(int)offset,(int)(signed char)*(treeptr-1),
         (int)tree[level][offset]);
   }

   return treeptr;


 case 2:         /* 10XXXXXX -> 00000010 go right code, skip left */
   if (DEBUG2-->0)
   {
    printf("level %i, right at %i, i=%i, o=%i  - pruned left,"
         " going right  \n",
         level,(int)offset,(int)(signed char)*(treeptr-1),
         (int)tree[level][offset]);
   }

   if(level>1)               /* continue */
   {
    treeptr = buildtree(tree,treevector,size,treeptr,
                  level-1,2*(offset)+1);
   }
   return treeptr;


 case 1:         /* 01XXXXXX -> 00000001 go left code, skip right */
   if (DEBUG2-->0)
   {
    printf("level %i, left at  %i, i=%i, o=%i - prune right,"
         " going left  \n",
         level,(int)offset,(int)(signed char)*(treeptr-1),
         (int)tree[level][offset]);
   }

   /*go left if level>1 */
   if(level>1)
   {
    treeptr = buildtree(tree,treevector,size,treeptr,level-1,
                  2*(offset));
```

```
        }

        if (DEBUG2-->0) printf("level %i, right at %i skipped\n",level,
                        offset);
        return treeptr;


      default:      /* 00XXXXXX -> 00000000 no code, process normally */
      /*go left if level>1) */

      if(level>1)
      {

        if (DEBUG2-->0) printf("level %i, left at  %i, i=%i, o=%i\n",
              level,offset,(int)((signed char)*(treeptr-1)),
              (int)((float)tree[level][offset]));

        treeptr = buildtree(tree,treevector,size,treeptr,
                        level-1,2*(offset));

        /* and then do the right side */


        if (DEBUG2-->0) printf("level %i, right at %i\n",
              level,offset);

       treeptr = buildtree(tree,treevector,size,treeptr,
                        level-1,2*(offset)+1);
      }
      else
      {
        if (DEBUG2-->0) printf("level %i, done  at %i, i=%i, o=%i\n",
                level,offset,(int)((signed char)*(treeptr-1)),
                (int)((float)tree[level][offset]));
      }
   }
   return treeptr;
}


/*************************************************************************/
void unzerotree(float *vector,signed char *treevector,int size,int levels)
{
  int i, maxlevel;
  float **loc;
  float **tree;
  signed char *treeptr;
  int offset = 0;

  printf("rebuilding tree pointers\n");

  i=0;
  while(1<<i<size) maxlevel=++i;
  loc = getvect(maxlevel+1,sizeof(int));
```

```
  tree = getvect(maxlevel+1,sizeof(int));

  printf("maxlevel is %i, levels to restore are %i\n",maxlevel,levels);

  if(levels>=maxlevel)
  {
    printf("too many levels requested\n");
    exit(0);
  }

  tree[maxlevel] = vector;
  loc[maxlevel] = vector;

  for (i=0;i<maxlevel;i++) /*set up pointers 0 = end, level+1 = begining */ {
   tree[i] = &vector[size>>i] ;
   loc[i] = tree[i];
  }

  if (DEBUG) printf("vector address = %i\n",(int)vector);
  for (i=maxlevel;i>0;i--)
  {
   printf("  loc   %i offset %i contains %i floats",
         i,loc[i]-loc[maxlevel],loc[i-1]-loc[i]);
   if (DEBUG) printf(", starts at addr %i",(int) loc[i]);
   printf("\n");
  }

  treeptr = treevector;

  i=0;
  while (&vector[i]< tree[levels])
  {
   treeptr = decode(treeptr,&vector[i]);
   if(DEBUG) printf("vect = %6.2f \n",vector[i]);
   i++;
  }

  printf("%i tokens copied\n",i);
  offset=0;

  /* while there is still input*/
  while(offset<=tree[levels]-tree[maxlevel])
  {
   treeptr = buildtree(tree,treevector,size,treeptr,levels,offset++);
  }
}


/*************************************************************************/
void main(int argc, char *argv[])
{
  int fsize;
  int xdim=0, ydim=0, size=0;
  float *vector;
```

```
float *headvector;
signed char *treevector;
float flevelq, flevelzt;
char string[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

setoptions(argc,argv);
infile = fopen(argv[1],"rb");
if(infile == NULL)
{
 printf("File open failure\n");
 exit(0);
}
strcpy(string,argv[1]);
if (direction>0) strcat(string,".zt2");
else strcat(string,".tz2");
outfile = fopen(string,"wb");
if(outfile== NULL)
{
 printf("File open failure\n");
 exit(0);
}
size = filesize(infile);
if (hsize==-1)  /* no header specified */
{
  fread(&flevelq,sizeof(float),1,infile);  /*levels processed by quant*/
  hsize = (flevelq+info)*2;
}

if(direction>0) size = ((size-1)/4)-hsize;
else size -= (hsize+info);

headvector =getvect(hsize,sizeof(float));
treevector =getvect(size*2,sizeof(signed char));

if (direction==1)
{
  vector = getvect(size,sizeof(float));
  fwrite(&flevelq,sizeof(float),info,outfile);/*levels processed by quant*/
  flevelzt = (float)level;
  fwrite(&flevelzt,sizeof(float),1,outfile);/*levels process by zt */
  fread(headvector,sizeof(float),hsize,infile);    /*get data vector*/
  fwrite(&size,sizeof(float),1,outfile);
  printf("saving header of size %i\n",hsize);
  fwrite(headvector,sizeof(float),hsize,outfile);  /*save data vector*/
  fread(vector,sizeof(float),size,infile);         /*get image vector */
  zerotree(vector,treevector,size,level);
  printf("output file write of %i\n",
    fwrite(treevector,sizeof(signed char),outcount,outfile));
  printf("%s = %i tokens, converted to %i char\n",argv[1],size,outcount);
  printf("%i long codes output\n",countlong);
  printf("output stop = %i, left = %i, right = %i, spec = %i, total code tokens %i\n",

count126,count127,countm127,countspec,count126+count127+countm127+countspec);
  printf("token overhead %3.2f%%\n",
```

```
        100.0*((float)(count126+count127+countm127+countspec)/(float)outcount));
  }
  else
  {
   fread(&flevelzt,sizeof(float),1,infile);/*level processed by zt */
   level = (int)flevelzt;
   fread(&fsize,sizeof(float),1,infile);
   vector = getvect(fsize,sizeof(float));
   fwrite(&flevelq,sizeof(float),info,outfile);
   fread(headvector,sizeof(float),hsize,infile);    /*get data vector*/
   printf("saving header of size %i\n",hsize);
   fwrite(headvector,sizeof(float),hsize,outfile);  /*save data vector*/
   fread(treevector,sizeof(signed char),size,infile);
   printf("reading in %i data tokens \n",size);
   unzerotree(vector,treevector,fsize,level);
   printf("%s = %i tokens, converted to %i floats\n",argv[1],size,outcount);
   printf("output file write of %i\n",
     fwrite(vector,sizeof(float),fsize,outfile));
  }

  free(vector);
  free(treevector);
  fclose(infile);
  fclose(outfile);
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: ZT.C

PURPOSE:  Builds/recovers a zero subtree from a wavelet
transform file

Input file is a binary float file
output file is a char file

Expects a data header of 1 + 2*levels processed (floats)

FEATURES: intelligent selection of data nodes to keep, uses 8 bit stop tolens

CONSTRAINTS:  no file size detection routines

-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define DEBUG  0
#define DEBUG2 0

/*********************************************************
GLOBALS
*********************************************************/
FILE *infile, *outfile;
int outcount = 0;
int count126 = 0;
int count127 = 0;
int countm127= 0;
int threshold1 = 0;
int threshold2 = 0;
int lookahead = 2;
int hsize = -1;
int level = 0;
int direction = 1;
int info = 1;
int dbgcount=100;
int dbg=0;

/*********************************************************/
void setoptions(int argc,char *argv[])
{
 int i;

 if(argc < 4)
 {
  printf("Zero Tree - zero tree for a wavelet transform file  \n");
```

```
    printf("This program is for float format files \n");
    printf("Usage: pgm filename LEVELS DIR [options]\n");
    printf("DIR 1 forward -1 backward      \n");
    printf("LEVELS levels to process   \n");
    printf("options:                   \n");
    printf("-T# primary   threshold { 0 default }    \n");
    printf("-t# secondary threshold { 0 default }  \n");
    printf("-a# lookahead levels    { 2 default }\n");
    printf("-h# size of header { checks file by default } \n");
    exit(0);
  }

  for (i=4;i<argc;i++)
  {
   switch (argv[i][1])
   {
     case 'a':lookahead = atoi(&argv[i][2]);
              break;
     case 'T':threshold1 = atoi(&argv[i][2]);
              break;
     case 't':threshold2 = atoi(&argv[i][2]);
              break;
     case 'h':hsize = atoi(&argv[i][2]);
              info = 0;
              break;
   }
  }

  level = atoi(argv[2]);
  direction = atoi(argv[3]);
  if (!abs(direction)==1) direction= 1;
  if (direction!=1) printf("levels = %i, direction %i\n",level,direction);
  else printf("levels = %i, direction %i, lookahead = %i, thresh1 = %i,"
           " thresh2 = %i\n",level,direction,lookahead,threshold1,
           threshold2);
}

/*********************************************************/
void * getvect(int n,int size)
{
  int i;
  void *vectptr;

  vectptr = (void *) calloc(n,size);
  if(vectptr == NULL) {
    printf("Error: dynamic memory fail\n");
    exit(0);
  }
  return(vectptr);
}

/*********************************************************/
/* file size determinator */
int  filesize(FILE *stream)
```

```
{
  int  curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}


/*************************************************************************/
signed char *encode(signed char *treeptr,float *input)
{
  int lowbyte=0xff;

  if(abs((int)*input)<126) /* -128,-127,+127,126 are reserved codes*/
  {
    *treeptr++ = (signed char)*input;
    outcount++;
  }
  else if (abs((int)*input)<=32895)
  {
    *treeptr++=-128;
    *treeptr++= (signed char)((int)*input>>8);
    *treeptr++= (unsigned char)((int)*input&lowbyte);
    outcount+=3;
    if(DEBUG) printf("decomp %i into %i + %i\n",(int)*input,
                (int)*(treeptr-2),(int)*(treeptr-1));
  }
  else
  {
    printf("ABORTED: range error exceeded on abs calc = %i\n",(int)*input);
    fwrite(&treeptr[-outcount],sizeof(signed char),outcount,outfile);
    printf("%i token saved in outfile\n",outcount);
    exit(0);
  }
  return treeptr;
}


/*************************************************************************/
int threshcheck(float *loc[],int level,int depth,int retval)
{
  int i, chkval;

  chkval = threshold1+(threshold2*depth);

  for (i=0;i<depth;i++)  /*left side*/
  {
    if(abs(loc[level][i])>chkval) retval|=2;
  }
  for (i=depth;i<depth*2;i++) /* right side*/
  {
    if(abs(loc[level][i])>chkval) retval|=1;
```

```
 }
 if (retval==3) return retval; /* both sides signif, stop */
 else if (depth<lookahead&&level>1)
    return threshcheck(loc,level-1,depth+1,retval);
 else return retval;
}


/*************************************************************************/
signed char *treetraverse(float *vector,signed char *treeptr, float *loc[],
                  float *tree[],int level)
{
 int i;
 int nextlevel;
 int initval =0;
 int initdepth =1;

 /*encode left node and subtree if not zero*/

 if (abs(*loc[level])<=threshold1&&level>1)
 {                      /* need to see if anything sig is below*/
  switch(threshcheck(loc,level-1,initdepth,initval))
  {
   case 0: /* both subtrees insignificant */
      if (DEBUG2)
      {
       printf("level %i, stop at  %i, i=%i, o=126  - both pruned, "
            "going back\n", level,(int)(loc[level]-tree[level]),
             (int)(signed char)*loc[level]);
      }
      *treeptr++ = 126;
      count126++;
      outcount++;
      nextlevel = level;      /*update subtree pointers 2 ea level*/
      for(i=nextlevel;i>0;i--)
      {
       loc[i] += 1<<(nextlevel-i);
      }
      return treeptr;

   case 1: /* left subtree insignificant, process right*/
      if (DEBUG2)
      {
       printf("level %i, right at %i, i=%i, o=127 -  pruned left, "
            "going right \n", level,(int)(loc[level]-tree[level]),
            (int)(signed char)*loc[level]);
      }
      *treeptr++ = 127;
      count127++;
      outcount++;

      /*update subtree pointers, before doing right */
      loc[level]++;           /* consume input */
```

```
        nextlevel = level-1;
        for(i=nextlevel;i>0;i--)
        {
          loc[i] += 1<<(nextlevel-i);
        }
        treeptr = treetraverse(vector,treeptr,loc,tree,level-1);
        return treeptr;

      case 2: /* right subtree insignificant, process left*/
        if (DEBUG2)
        {
          printf("level %i, left at  %i, i=%i, o=-127 - pruned right, "
                 "going left  \n", level,(int)(loc[level]-tree[level]),
                (int)(signed char)*loc[level]);
        }
        *treeptr++ = -127;
        outcount++;
        countm127++;
        treeptr = treetraverse(vector,treeptr,loc,tree,level-1);

        if (DEBUG2) printf("level %i, right at %i skipped\n",level,
                        (int)(loc[level]-tree[level]));
        /*update subtree pointers, after left done */
        nextlevel = level-1;
        for(i=nextlevel;i>0;i--)
        {
          loc[i] += 1<<(nextlevel-i);
        }
        loc[level]++;
        return treeptr;

      /* case 3: both sides are significant fall through and process */
    }
  }

  /* or else process normally */

  treeptr = encode(treeptr,loc[level]);

  if(level>1)
  {

    if (DEBUG2) printf("level %i, left at  %i, i=%i, o=%i\n",
        level,loc[level]-tree[level],(int)((signed char)*loc[level]),
        (int)((signed char)*(treeptr-1)));

    treeptr = treetraverse(vector,treeptr,loc,tree,level-1);

    /*encode right subtree if level>1*/
/*
  if (DEBUG2) printf("level %i, right at %i, i=%i, o=%i\n",
        level,loc[level]-tree[level],(int)((signed char)*loc[level]),
        (int)((signed char)*(treeptr-1)));
*/
```

```
    if (DEBUG2) printf("level %i, right at %i\n",level,
                    loc[level]-tree[level]);

    treeptr = treetraverse(vector,treeptr,loc,tree,level-1);
   }
   else
   {
    if (DEBUG2) printf("level %i, done  at %i, i=%i, o=%i\n",
         level,loc[level]-tree[level],(int)((signed char)*loc[level]),
         (int)((signed char)*(treeptr-1)));
   }
   loc[level]++; /* update location */
   return treeptr;
}


/*************************************************************************/
void zerotree(float *vector,signed char *treevector,int size,int levels)
{
  int i, maxlevel;
  float **loc;
  float **tree;
  signed char *treeptr;

  printf("setting up tree pointers\n");

  i=0;
  while(1<<i<size) maxlevel=++i;
  loc = getvect(maxlevel+1,sizeof(int));
  tree = getvect(maxlevel+1,sizeof(int));

  printf("maxlevel is %i, levels requested are %i\n",maxlevel,levels);

  if(levels>maxlevel)
  {
    printf("too many levels requested\n");
    exit(0);
  }

  tree[maxlevel] = vector; /* should add test for 1> 2**levels */
  loc[maxlevel] = vector;

  for (i=0;i<maxlevel;i++) /* set up pointers 0 = end, maxlevel = start*/
  {
   tree[i] = &vector[size>>i] ;
   loc[i] = tree[i];
  }

  if(DEBUG)printf("vector address = %i\n",(int)vector);

  for (i=maxlevel;i>0;i--)
  {
   printf("  loc   %i offset %i contains %i floats",
         i,loc[i]-loc[maxlevel],loc[i-1]-loc[i]);
```

```
    if(DEBUG) printf(", starts at addr %i",(int)loc[i]);
    printf("\n");
   }

   treeptr = treevector;

   i=0;
   while (&vector[i]< tree[levels])
   {
    treeptr = encode(treeptr,&vector[i]);
    if(DEBUG) printf("vect = %6.2f gives c =%i \n",
                vector[i],(int)treeptr[-1]);
    i++;
   }

   printf("%i tokens copied\n",i);

   while (loc[levels]<tree[levels-1]) /* begin recursive tree search call */
   {
    treeptr = treetraverse(vector,treeptr,loc,tree,levels);
   }

}


/*************************************************************************/
signed char *decode(signed char *treeptr,float *outputloc)
{
 int temp;

 if (*treeptr==-128)   /* rebuild integer value */
 {
  treeptr++;
  temp = (int)((signed char)*treeptr++);
  temp <<=8;
  temp += (int)((unsigned char)*treeptr++);
  *outputloc = (float)temp;
  outcount++;
  if (DEBUG) printf("rebuild %i from %i + %i\n",
       (int)*outputloc,(int)*(treeptr-2),(int)*(treeptr-1));
 }
 else
 {
  *outputloc=(float)*treeptr++;
  outcount++;
 }
 return treeptr;
}


/*************************************************************************/
signed char *buildtree(float *tree[], signed char *treevector,int size,
          signed char *treeptr, int level,int offset)
{
```

```
signed char *treelast;

if (offset>=tree[level-1]-tree[level])
{
 printf("level %i ,size %i, index %i completed\n",
        level,tree[level-1]-tree[level],offset);
 return treeptr;
}
switch (*treeptr)
{
 case 126:
    if (DEBUG2)
    {
     printf("level %i, stop at  %i, i=%i, o=0  - both pruned, going back  \n",
          level,(int)offset,(int)(signed char)*treeptr);
    }
    /* consume token */
    outcount++;
    return ++treeptr;


 case 127:
    if (DEBUG2)
    {
     printf("level %i, right at %i, i=%i, o=0  - pruned left, going right  \n",
          level,(int)offset,(int)(signed char)*treeptr);
    }
    outcount++;
    treeptr++;
    /*if input exhasted quit */
    if (offset>=tree[level-1]-tree[level])
    {
     printf("level %i ,size %i, index %i completed\n",
          level,tree[level-1]-tree[level],offset);
     return treeptr;
    }


    if(level>1)              /* continue */
    {
     treeptr = buildtree(tree,treevector,size,treeptr,
                    level-1,2*(offset)+1);
    }
    return treeptr;


case -127:       /* skip right*/
   if (DEBUG2)
   {
    printf("level %i, left at  %i, i=%i, o=0 - prune right, going left  \n",
         level,(int)offset,(int)(signed char)*treeptr);
   }
   outcount++;
   treeptr++;
```

```
    /*if input exhasted quit */
    if (offset>=tree[level-1]-tree[level])
    {
     printf("level %i ,size %i, index %i completed\n",
          level,tree[level-1]-tree[level],offset);
     return treeptr;
    }

    /*go left if level>1 */
    if(level>1)
    {
     treeptr = buildtree(tree,treevector,size,treeptr,level-1,
                  2*(offset));
    }

    if (DEBUG2) printf("level %i, right at %i skipped\n",level,
                offset);
    return treeptr;


  default:      /* process normally */
    /*go left if level>1) */

    treeptr = decode(treeptr,&tree[level][offset]);

    if(level>1)
    {

     if (DEBUG2) printf("level %i, left at  %i, i=%i, o=%i\n",
          level,offset,(int)((signed char)*(treeptr-1)),
          (int)((float)tree[level][offset]));

     treeptr = buildtree(tree,treevector,size,treeptr,
                  level-1,2*(offset));

     /* and then do the right side */

     /*if input exhasted quit */
     if (offset>=tree[level-1]-tree[level])
     {
      printf("level %i ,size %i, index %i completed\n",
           level,tree[level-1]-tree[level],offset);
      return treeptr;
     }

     if (DEBUG2) printf("level %i, right at %i\n",
          level,offset);

     treeptr = buildtree(tree,treevector,size,treeptr,
                  level-1,2*(offset)+1);
    }
    else
    {
     if (DEBUG2) printf("level %i, done  at %i, i=%i, o=%i\n",
```

```
                level,offset,(int)((signed char)*(treeptr-1)),
                (int)((float)tree[level][offset]));
        }
    }
    return treeptr;
}


/*************************************************************************/
void unzerotree(float *vector,signed char *treevector,int size,int levels)
{
    int i, maxlevel;
    float **loc;
    float **tree;
    signed char *treeptr;
    int offset = 0;

    printf("rebuilding tree pointers\n");

    i=0;
    while(1<<i<size) maxlevel=++i;
    loc = getvect(maxlevel+1,sizeof(int));
    tree = getvect(maxlevel+1,sizeof(int));

    printf("maxlevel is %i, levels to restore are %i\n",maxlevel,levels);

    if(levels>=maxlevel)
    {
      printf("too many levels requested\n");
      exit(0);
    }

    tree[maxlevel] = vector;
    loc[maxlevel] = vector;

    for (i=0;i<maxlevel;i++) /*set up pointers 0 = end, level+1 = begining */ {
      tree[i] = &vector[size>>i] ;
      loc[i] = tree[i];
    }

    if (DEBUG) printf("vector address = %i\n",(int)vector);
    for (i=maxlevel;i>0;i--)
    {
      printf("  loc   %i offset %i contains %i floats",
            i,loc[i]-loc[maxlevel],loc[i-1]-loc[i]);
      if (DEBUG) printf(", starts at addr %i",(int) loc[i]);
      printf("\n");
    }

    treeptr = treevector;

    i=0;
    while (&vector[i]< tree[levels])
```

```
  {
    treeptr = decode(treeptr,&vector[i]);
    if(DEBUG) printf("vect = %6.2f \n",vector[i]);
    i++;
  }

  printf("%i tokens copied\n",i);

  offset=0;

  /* while there is still input*/

  while(offset<=tree[levels]-tree[maxlevel])
  {
    treeptr = buildtree(tree,treevector,size,treeptr,levels,offset++);
  }

}


/*********************************************************/
/*********************************************************/
/*********************************************************/
/* MAIN                                                  */
/*********************************************************/

void main(int argc, char *argv[])
{
  int fsize;
  int xdim=0, ydim=0, size=0;
  float *vector;
  float *headvector;
  signed char *treevector;
  float flevelq, flevelzt;
  char string[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

  setoptions(argc,argv);

  infile = fopen(argv[1],"rb");
  if(infile == NULL)
  {
   printf("File open failure\n");
   exit(0);
  }
  strcpy(string,argv[1]);


  if (direction>0) strcat(string,".zt");
  else strcat(string,".tz");

  outfile = fopen(string,"wb");
  if(outfile== NULL)
  {
   printf("File open failure\n");
```

```
 exit(0);
}

size = filesize(infile);
if (hsize==-1)  /* no header specified */
{
 fread(&flevelq,sizeof(float),1,infile);  /*levels processed by quant*/
 hsize = (flevelq+info)*2;
}
if(direction>0) size = ((size-1)/4)-hsize;
else size -= (hsize+info);
headvector =getvect(hsize,sizeof(float));
treevector =getvect(size*2,sizeof(signed char));
if (direction==1)
{
 vector = getvect(size,sizeof(float));
 fwrite(&flevelq,sizeof(float),info,outfile);/*levels processed by quant*/
 flevelzt = (float)level;
 fwrite(&flevelzt,sizeof(float),1,outfile);/*levels process by zt */
 fread(headvector,sizeof(float),hsize,infile);   /*get data vector*/
 fwrite(&size,sizeof(float),1,outfile);
 printf("saving header of size %i\n",hsize);
 fwrite(headvector,sizeof(float),hsize,outfile);  /*save data vector*/
 fread(vector,sizeof(float),size,infile);        /*get image vector */
 zerotree(vector,treevector,size,level);
 fwrite(treevector,sizeof(signed char),outcount,outfile);
 printf("%s = %i tokens, converted to %i char\n",argv[1],size,outcount);
 printf("output 126 = %i, 127 = %i, -127 = %i total code tokens %i\n",
       count126,count127,countm127,count126+count127+countm127);
 printf("token overhead %3.2f\n",
        100.0*((float)(count126+count127+countm127)/(float)outcount));
}
else
{
 fread(&flevelzt,sizeof(float),1,infile);/*level processed by zt */
 level = (int)flevelzt;
 fread(&fsize,sizeof(float),1,infile);
 vector = getvect(fsize,sizeof(float));
 fwrite(&flevelq,sizeof(float),info,outfile);
 fread(headvector,sizeof(float),hsize,infile);   /*get data vector*/
 printf("saving header of size %i\n",hsize);
 fwrite(headvector,sizeof(float),hsize,outfile);  /*save data vector*/
 fread(treevector,sizeof(signed char),size,infile);
 unzerotree(vector,treevector,fsize,level);
 fwrite(vector,sizeof(float),fsize,outfile);
 printf("%s = %i tokens, convert to %i floats, expanded to %i\n",argv[1],size,outcount,fsize);
}

free(vector);
free(treevector);
fclose(infile);
fclose(outfile);
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: WAVE.C

PURPOSE: Perform a linear orthogonal transform on a floating point data file

Input file is a binary float file
output file is a float file

FEATURES: Has several options to control the selection of filters and the
number of levels processed, allows snaking of the input on the row length
to minimize the edge effect of the image on the high band pass

-------------------------------------------------------
-------------------------------------------------------*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MAXSIZE 262145


/*************************************************************
GLOBALS
*************************************************************/

struct controlblock {
  char infile[81];
  char outfile[81];
  char root[81];
  char extension[5];
  int  filter;
  int  compress;
  char action;
  int  level;
  int  shift;
  double normal;
  int  rotate;
  int  snake;
};
/* daubechies filters 2 - 20 */
double filtcoef[11][20] =
{
 { 1.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
   0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
   0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
   0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
   0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000  },
 /* daub 2 */
 { 0.707106781187, 0.707106781187, 0.000000000000, 0.000000000000,
   0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
```

```
  0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000,  0.000000000000, 0.000000000000 },
/* daub 4 */
{ 0.482962913145, 0.836516303738, 0.224143868042,-0.129409522551,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 6 */
{ 0.332670552950, 0.806891509311, 0.459877502118,-0.135011020010,
 -0.085441273882, 0.035226291882, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 8 */
{ 0.230377813309, 0.714846570553, 0.630880767930,-0.027983769417,
 -0.187034811719, 0.030841381836, 0.032883011667,-0.010597401785,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 10 */
{ 0.160102397974, 0.603829269797, 0.724308528438, 0.138428145901,
 -0.242294887066,-0.032244869585, 0.077571493840,-0.006241490213,
 -0.012580751999, 0.003335725285, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 12 */
{ 0.111540743350, 0.494623890398, 0.751133908021, 0.315250351709,
 -0.226264693965,-0.129766867567, 0.097501605587, 0.027522865530,
 -0.031582039318, 0.000553842201, 0.004777257511,-0.001077301085,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 14 */
{ 0.077852054085, 0.396539319482, 0.729132090846, 0.469782287405,
 -0.143906003929,-0.224036184994, 0.071309219267, 0.080612609151,
 -0.038029936935,-0.016574541631, 0.012550998556, 0.000429577973,
 -0.001801640704, 0.000353713800, 0.000000000000, 0.000000000000,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 16 */
{ 0.054415842243, 0.312871590914, 0.675630736297, 0.585354683654,
 -0.015829105256,-0.284015542962, 0.000472484574, 0.128747426620,
 -0.017369301002,-0.044088253931, 0.013981027917, 0.008746094047,
 -0.004870352993,-0.000391740373, 0.000675449406,-0.000117476784,
  0.000000000000, 0.000000000000, 0.000000000000, 0.000000000000 },
/* daub 18 */
{ 0.038077947364, 0.243834674613, 0.604823123690, 0.657288078051,
  0.133197385825,-0.293273783279,-0.096840783223, 0.148540749338,
  0.030725681479,-0.067632829061, 0.000250947115, 0.022361662124,
 -0.004723204758,-0.004281503682, 0.001847646883, 0.000230385764,
 -0.000251963189, 0.000039347320, 0.000000000000, 0.000000000000 },
/* daub 20 */
{ 0.026670057901, 0.188176800078, 0.527201188932, 0.688459039454,
  0.281172343661,-0.249846424327,-0.195946274377, 0.127369340336,
```

```
   0.093057364604,-0.071394147166,-0.029457536822, 0.033212674059,
   0.003606553567,-0.010733175483, 0.001395351747, 0.001992405295,
  -0.000685856695,-0.000116466855, 0.000093588670,-0.000013264203 }
};


/*********************************************************/
/* openfile - open file and check for errors */
/*********************************************************/

FILE *openfile(char *file, char *opt)
{
 FILE *ptr;

 if((ptr = fopen(file, opt)) == NULL)
 {
  fprintf(stderr, "Can not open the file %s\n", file);
  exit(1);
 }
 return(ptr);
}

/*********************************************************/
/* loadfile - loads in a data file of floating point numbers */
/*********************************************************/

long loadfile(float *data, struct controlblock control)
{
 FILE *in;
 int i,j;
 long size;
 int width,dblwidth,hlfwidth,span;
 long len;
 float tmp;

 /* open the input file */

 printf("loading %s\n",control.infile);

 {
  in = openfile(control.infile, "rb");
  fseek(in, 0L, 2);
  len = ftell(in);
  fseek(in, 0L, 0);
  if(len%sizeof(float))
  {
   fprintf(stderr, "Error in reading input file.\n");
   exit(1);
  }
  else
  {
   size = len/sizeof(float);
   if(size > MAXSIZE)
   {
```

```
      fprintf(stderr, "Input file is too large.\n");
      exit(1);
     }
     else
     {
      if(fread(data, sizeof(float), size, in) != size)
      {
       fprintf(stderr, "Error reading input file.\n");
       exit(1);
      }
     }
    }
    if (control.action!='i'&&control.snake>0)  /* reverse every other row */
    {
     width = control.snake;
     printf("snake option with width %i used\n",width);
     dblwidth = width*2;
     hlfwidth = width/2;
     for (j=width;j<size;j+=dblwidth)
     {
      span = j+width-1;
      for(i=0;i<hlfwidth;i++)
      {
       tmp = data[j+i];
       data[j+i] = data[span-i];
       data[span-i] = tmp;
      }
     }
    }
  }

  fclose(in);
  return(size);
}


/*********************************************************/
/* extendedhex - This routine takes an integer between 0 and 36 and converts
      it into a character that goes from 0-9,A-Z.    */
/*********************************************************/

char extendedhex(int value)
{
  if((value > 35) || (value < 0))
  {
    fprintf(stderr, "Value too big to specify the file extension properly.\n");
    exit(1);
  }
  if(value < 10) return('0' + value);
  else return('a' + value - 10);
}

/*********************************************************/
/* savefile - This routine saves results in output using the coined extension */
```

```
/*********************************************************/

void savefile(float *data,int level,long length,struct controlblock control)
{
 FILE *out;
 int i,j,size;
 int width,dblwidth,hlfwidth,span;
 float tmp;


 /* construct the output name */
 if(strlen(control.root) != NULL)
 {
  control.extension[0] = '.';

  /* use the action type for the first byte */
  if(control.action == 's')
  {
   control.extension[1] = extendedhex(control.level);
  }
  else
  {
   control.extension[1] = control.action;
  }

  /* use the wavelet type for the second byte */
  control.extension[2] = extendedhex(2*control.filter);

  /* use the level of the decomposition for the third byte */
  control.extension[3] = extendedhex(level);
  control.extension[4] = 0x00;
  strcpy(control.outfile, control.root);
  strcat(control.outfile, control.extension);
 }

 if (control.action=='i'&&control.snake>0)  /* reverse every other row */
 {
  width = control.snake;
  printf("snake option with width %i used\n",width);
  dblwidth = width*2;
  hlfwidth = width/2;
  for (j=width;j<length;j+=dblwidth)
  {
   span = j+width-1;
   for(i=0;i<hlfwidth;i++)
   {
    tmp = data[j+i];
    data[j+i] = data[span-i];
    data[span-i] = tmp;
   }
  }
 }

 /* open the output file and save */
```

```
  printf("saving %s\n",control.outfile);
  out = openfile(control.outfile, "wb");
  if(fwrite(data, sizeof(float), length, out) != length)
  {
    fprintf(stderr, "Error writing ouput file.\n");
    exit(1);
  }
  fclose(out);
}

/*********************************************************/
/* getvector - memory allocation, error check and initialize */
/*********************************************************/

void *getvector(int size, long length)
{
  char *ptr;
  long i;

  ptr = (char *)malloc(size*length);
  if(ptr == NULL)
  {
    fprintf(stderr, "Error in allocating memory.\n");
    exit(1);
  }

  /* initialize */
  for(i=0; i<size*length; i++) ptr[i] = 0;
  return((void *)ptr);
}

/*********************************************************/
/* copyvect - copy one vector to another, pad left on positive, right
   on negative */
/*********************************************************/

void copyvect(float *out, float *in, long length, int overrun)
{
  long i;

  printf("vector copy, size %i\n",length);

  if(overrun < 0)
  {
    overrun *= -1;

    for(i=0; i<length; i++) out[overrun + i] = in[i];

    /* i%length allows multiple repetitions if input shorter than overrun */
    for(i=0; i<overrun; i++)
      out[overrun-1-i] = out[overrun+length-1-i%length];
  }
  else
  {
```

```
    for(i=0; i<length; i++) out[i] = in[i];
    /* continually repeat while necessary */
    for(i=0; i<overrun; i++) out[length+i] = out[i%length];
  }
}


/*********************************************************/
/* minsize - calculate a minimum length that is power of 2 and
   wil contain the data */
/*********************************************************/

long minsize(int *iterations, long number)
{
  int loops = 0;
  long pow2 = 1;

  while(1)
  {
    if(pow2 >= number)
    break;
    pow2 <<= 1;
    loops++;
  }

  /* the analysis can't exceed the length of the data */
  if(*iterations == 0) *iterations = loops;
  else if(*iterations > loops)
  {
    fprintf(stderr, "Data not long enough for the number of iterations requested\n");
    fprintf(stderr, "Max # of iterations possible is %d.\n", loops);
    exit(1);
  }
  return(pow2);
}


/*********************************************************/
/* buildfilter - sets up wavelet coefficients for transform,
   if filt is negative then sets up reconstruction coefficients */
/*********************************************************/

void buildfilter(double *even, double *odd, int filter)
{
  int i;
  if(filter > 0)
  {
    for(i=0; i<2*filter; i++)
    {
      /* low pass */
      even[i] = filtcoef[filter][i];
      /* high pass */
      odd[i] = filtcoef[filter][2*filter - i -1];
      if(i%2) odd[i] *= -1;
```

```
    }
   }
   else
   {
    filter *= -1;
    for(i=0; i<filter; i++)
    {
     /* inverse even index */
     even[i*2] = filtcoef[filter][filter*2 - 2*(i+1)];
     even[i*2+1] = filtcoef[filter][2*i+1];

     /* inverse odd index */
     odd[i*2] = filtcoef[filter][filter*2 - (2*i+1)];
     odd[i*2+1] = -filtcoef[filter][i*2];
    }
   }
}


/*********************************************************/
/* wavelet - performs wavelet decomposition, returns high frequency component
 in first half of the vector and low frequency in last half of the vector.  */
/*********************************************************/

void wavelet(float *buf, long length, int width, double *evenfilter,
          double *oddfilter)
{
 long i;
 int j;
 int times;
 float even, odd;


 printf("transforming, width %i\n",width);

 /* begin the loop through the data */
 for(i=0; i<length; i += 2)
 {
  even = 0.;
  odd = 0.;

  for(j=0; j<width; j++)
  {
   even += evenfilter[j]*buf[i+j];
   odd += oddfilter[j]*buf[i+j];
  }
  buf[i] = even;
  buf[i+1] = odd;
 }
}


/*********************************************************/
/* decimate - permutes the vector into a second, decimate if length positive
```

```
  or interpolate if negative, neg overrun right to left, pos overrun left to
  right */
/*********************************************************/

void decimate(float *out, float *in, long length, int overrun)
{
 long i;
 long halflength;
 int offset;


 printf("vector permute, length %i\n",length);

 if(length < 0) halflength = -length/2;
 else halflength = length/2;

 if(overrun < 0) offset = -overrun;
 else offset = 0;

 if(length < 0)
 {
  /* decimate */
  length *= -1;
  for(i=0; i<halflength; i++)
  {
   /* copy the even index - low pass result */
   out[offset+i*2] = in[i];

   /* copy the odd index - high pass result */
   out[offset+i*2 + 1] = in[i+halflength];
  }
 }
 else
 {
  /* split the alternate values up */
  for(i=0; i<halflength; i++)
  {
   out[offset+i] = in[i*2];
   out[offset+i+halflength] = in[i*2+1];
  }
 }

 if(overrun < 0)
 {
  /* pad the left side */
  overrun *= -1;

  for(i=0;i<overrun;i++)
   out[overrun-1-i] = out[overrun+length-1-i%length];
 }
 else
 {
  /* pad the right side */
  for(i=0; i<overrun; i++) out[length+i] = out[i%length];
```

```
  }
}


/*********************************************************/
/* rotate - shifts the vector right or left based on the value of dir,
   left-over data values are wrapped to the other end */
/*********************************************************/

void rotate(float *data, long length, int shift, int dir)
{
  float bfs[20];  /* is good up to D40 filter */
  long i;

  /* if shift distance exceeds the length then reduce the shift */
  shift %= length;

  if(dir == 1)     /* rotate upward */
  {
    for(i=0;i<shift;i++)       bfs[i] = data[length-shift+i];
    for(i=0;i<length-shift;i++) data[length-i-1] = data[length-shift-i-1];
    for(i=0;i<shift;i++)       data[i] = bfs[i];
  }
  if(dir == -1)     /* rotate downward */
  {
    for(i=0;i<shift;i++)       bfs[i] = data[i];
    for(i=0;i<length-shift;i++) data[i] = data[i+shift];
    for(i=0;i<shift;i++)       data[length-shift+i] = bfs[i];
  }
}


/*********************************************************/
/* help - prints options list to the screen   */
/*********************************************************/
void help(void)
{
  printf("wavelet <input> (options)\t default options: -d4 -n2 -t\n");
  printf("Options:\n");
  printf("-o <output>\n\tOutput file root or whole name.\n");
  printf("-d#\tSpecify the wavelet transform d2 through d20.\n");
  printf("-n#\t1/2 Normalization. 1-On inverse, 2-Root on both, 3-On transform.\n");
  printf("-t#\tPerform the transform to # levels.\n");
  printf("-i#\tPerform the inverse transform to # levels of the pyramid.\n");
  printf("-w#\tSnake input data on row width given. \n");
  printf("-s#\tSame as -t# but save each level as a separate file.\n");
  printf("-l#\tPerform a low pass filter on the data transformed to the # level.\n");
  printf("-h#\tPerform a high pass filter on the data transformed to the # level.\n");
  printf("\t(Analyze to the highest level possible if # is not specified.)\n");
  printf("-r\tCorrelate coefficients at different levels by shifting results.\n");
}

/*********************************************************/
/* setcontrol - sets command line options in 'controlblock'*/
```

```
/*********************************************************/

struct controlblock setcontrol(int argc, char *argv[])
{
 int i;
 int level;
 char *ptr;
 struct controlblock control;

 /* set up the default values */
 strcpy(control.infile, "wavelet.dat");
 strcpy(control.outfile, "");
 strcpy(control.root, "");
 control.filter   = 2;
 control.compress = 0;
 control.normal   = 1.0;
 control.action   = 't';
 control.level    = 0;
 control.shift    = 0;
 control.rotate   = 0;
 control.snake    = 0;

 for(i=1; i<argc; i++)
 {
  if(argv[i][0] == '-')
  {
   switch(argv[i][1])
   {
    case '?':
          help();
          exit(0);
          break;
    case 'o':
          strcpy(control.outfile, argv[i+1]);
          i++;
          break;
    case '1':
          control.shift = 1;
          break;
    case 'n':
          level = atoi(&argv[i][2]);
          if ((level<1)||(level>3))
          {
           fprintf(stderr, "%s is not a valid option.\n", argv[i]);
           exit(1);
          }
          if (atoi(&argv[i][2])!=2)
           control.normal = 1.41421356237309504880;
          if (atoi(&argv[i][2])>2) control.normal = 1/control.normal;
          break;
    case 'r':
          control.rotate = 1;
          break;
    case 'd':
```

```
            control.filter = atoi(&argv[i][2]);
            if((control.filter%2 == 1) || (control.filter < 2) ||
              (control.filter > 20))
            {
              fprintf(stderr, "%s is not a valid option.\n", argv[i]);
              exit(1);
            }
            else control.filter /= 2;
            break;
      case 'c':
            control.compress = 1;
            break;
      case 'l':
            control.action = 'l';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 'h':
            control.action = 'h';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 'b':
            control.action = 'b';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 'k':
            control.action = 'k';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 'i':
            control.action = 'i';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 't':
            control.action = 't';
            level = atoi(&argv[i][2]);
            if (level>0&&level<22)  control.level = level;
            break;
      case 's':
            control.action = 's';
            level = atoi(&argv[i][2]);
            if (level>0) control.level = level;
            break;
      case 'w':
            control.snake = atoi(&argv[i][2]);
            break;
      default : fprintf(stderr, "invalid option = %d\n", argv[i]);
            exit(1);
    }
  }
```

```
    else
    {
      strcpy(control.infile, argv[i]);
    }
  }

  /* set up the root and extension of the output files */
  if(strlen(control.outfile) == 0)
  {
    strcpy(control.outfile, control.infile);
    ptr = strchr(control.outfile, '.');
    if(ptr)
    ptr[0] = 0x00;
    strcpy(control.root, control.outfile);
  }
  else
  {
    if(strchr(control.outfile, '.') == NULL) strcpy(control.root, control.outfile);
    else if(control.action == 's')
    {
       fprintf(stderr, "No output file extension allowed with option -s\n");
    }
  }

  return(control);
}

/*********************************************************/
/* MAIN                                                  */
/*********************************************************/

int main(int argc, char *argv[])
{
  long i;
  int j;
  float *data;
  float *buf;
  long num;
  long pow2;
  long width;
  long offset;
  struct controlblock control;
  double evenfilter[20];
  double oddfilter[20];
  int wvshft[11][2] = {{0,0},{0,0},{0,1},{0,2},{0,3},{1,3},
                       {1,4},{1,5},{1,6},{1,7},{1,8}};

  /* get command line options */
  if(argc < 2)
  {
    fprint("\nwavelet <input file> -(options)\n");
    fprint( "   Use -? for a help file.\n");
    exit(1);
  }
```

```
else
{
  control = setcontrol(argc, argv);
}

/* allocate the memory */
printf("starting program\n");
data = (float *)getvector(sizeof(float), MAXSIZE);
buf  = (float *)getvector(sizeof(float), MAXSIZE + 2*(control.filter-1));
/* load data */
num = loadfile(data, control);
/* determine the largest power of two that covers the data */
pow2 = minsize(&control.level, num);
printf("depth = %i, for file size %i, %i levels\n",pow2,num,control.level);
/* set up the filter */
buildfilter(evenfilter, oddfilter, control.filter);
/* apply normalization */
for(j=0; j<control.filter*2; j++)
{
  evenfilter[j] *= control.normal;
  oddfilter[j] *= control.normal;
}

/* if -1 is used shift the data one to the left */

if((control.shift==1)&&(control.action!='i')) rotate(data, pow2, 1, -1);

/* perform filter pass, save the high frequency component in lower
half of vector and the low frequency component in upper half */

width = pow2;
for(j=0; j<control.level; j++)
{
  /* don't decompose if only doing the inverse transform */
  if(control.action != 'i')
  {
    /* do the transform */
    copyvect(buf, data, width, 2*(control.filter-1));
    wavelet(buf, width, 2*control.filter, evenfilter, oddfilter);
    decimate(data, buf, width, 0);
    /* rotate the low and high frequency numbers to register properly */
    if(control.rotate)
    {
      rotate(data, width/2, wvshft[control.filter][0], 1);
      rotate(data+width/2, width/2, wvshft[control.filter][1], 1);
    }
  }

  /* shift to the upper half so the decomposition can be done again */
  width /= 2;
}

/* if only transform requested save and quit */
if(control.action == 't')
```

```
{
  savefile(data, control.level, pow2, control);
  exit(0);
}

if(control.action == 's')
{
  savefile(data, 0, width,control);
  offset = width;
  for(j=0; j<control.level; j++)
  {
    savefile(data + offset, j+1, offset, control);
    offset *= 2;
  }
  exit(0);
}

/* if filtering remove either the high or low frequency component  */
if((control.action == 'h') || (control.action == 'b'))
{
  for(i=0; i<width; i++) data[i] = 0.;
}

if(control.action == 'b')
{
  for(i=width*2; i<pow2; i++) data[i] = 0.;
}

if(control.action == 'k')
{
  for(i=width; i<width*2; i++)
  data[i] = 0.;
}

if(control.action == 'l')
{
  for(i=width; i<pow2; i++)
  data[i] = 0.;
}

/* set up the reconstruction filter */
buildfilter(evenfilter, oddfilter, -control.filter);

/* apply normalization */
for(j=0; j<control.filter*2; j++)
{
  evenfilter[j] /= control.normal;
  oddfilter[j] /= control.normal;
}

/* reconstruct the input from the transform results */
for(j=0; j<control.level; j++)
{
```

```
    /* expand to the next largest scale to decompose again */
    width *= 2;

    /* shift the low and high frequency vectors for reconstruction */

    if(control.rotate)
    {
      rotate(data, width/2, wvshft[control.filter][0], -1);
      rotate(data+width/2, width/2, wvshft[control.filter][1], -1);
    }

    /* perform the inverse transform */
    decimate(buf, data, -width, -2*(control.filter-1));
    wavelet(buf, width, 2*control.filter, evenfilter, oddfilter);
    copyvect(data, buf, width, 0);
  }

  /* if -1 is used shift the data one to the left */
  if(control.shift == 1) rotate(data, pow2, 1, 1);

  /* output result */
  savefile(data, control.level, num, control);
  printf("program complete\n");
  return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: QONLY.C

PURPOSE: Quantizes an input float vector from a wavelet
transform file

Input file is a binary float file
output file is a char file

Creates a data header of 1 + 2*levels processed (floats)
  Containing levels processed and mean and standard deviation of each
  level processed.

FEATURES: Uses a bin calculation routine that is based on 3 times the
standard deviation, puts 93% of the coefficients into 64 bins (+/-32)

-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <string.h>

#define DEBUG 1

FILE *infile, *outfile, *binfile;


/*****************************************************/
void * getvect(int n,int size)
{
  void *vectptr;

  vectptr = (void *) calloc(n,size);
  if(vectptr == NULL) {
    printf("Error: dynamic memory fail\n");
    exit(0);
  }
  return(vectptr);
}

/*****************************************************/
void quant(float *vector,float *headvector,int size,
          int stddev,int mean,int debug)
{
  float fstddev, fmean, frtsdv;
  int i=0;

  fstddev = (float)stddev/100;
  frtsdv = (fstddev>1) ? (float)sqrt((double)fstddev):1;
```

```
  fmean = (float) mean;
  mean *=100;

  headvector[0] = mean;
  headvector[1] = stddev;

  while (i<size)
  {
   vector[i] = (vector[i]-fmean)/frtsdv;
   i++;
  }
}

/*****************************************************/
void unquant(float *vector,float *headvector,int size,int debug)
{
 int i=0, stddev, mean=0, rebuilt;
 float fstddev, fmean, frtsdv, temp;

 mean = headvector[0];
 stddev = headvector[1];

 fstddev = (float)(stddev)/100.0;
 frtsdv = (fstddev>1) ? (float)sqrt((double)fstddev):1;
 fmean = (float) mean/100;
 printf("mean = %6.2f, stddev = %6.2f, root = %6.2f\n",fmean,fstddev,frtsdv);

 while (i<size)
 {
  vector[i]=(vector[i]*frtsdv)+fmean;
  i++;
 }
}

/*****************************************************/

void sectquant(float *vector,float *headvector, int size,int zeroout,int quantize,int debug)
{
 int i;
 double high=-10000000.0, low=10000000.0;
 int zeros=0, zrun=0, zcount=0, outsiders=0;
 double sum=0.0, squares=0.0, mean=0.0, var=0.0, stddev=0.0, sd4=0.0;
 double rtsdv=0.0;

 if (quantize==-1) unquant(vector,headvector,size,debug);

 for (i=0;i<size;i++)
 {
  high = (vector[i]>high) ? vector[i] : high;
  low  = (vector[i]<low ) ? vector[i] : low ;
  sum += vector[i];
  squares += (double)vector[i]*vector[i]/(double)size;
```

```
  }
  mean = sum/(double)size;
  var = squares - (mean*mean);
  stddev = sqrt(var);
  sd4 = 3*stddev;
  rtsdv = sqrt(stddev);  /* zero threshold, sqrt of std dev */


  for (i=0;i<size;i++)
  {
   if ((double)abs(vector[i])>sd4) outsiders++;
   if ((double)abs(vector[i])<rtsdv)
   {
    zeros++;
    if (quantllzeroout) vector[i]=0.0;
   }
   else if(/*(double)abs(vector[i])<sd4&&*/zeroout)
   {
    zeros++;
    vector[i]=0.0;
   }
   if ((double)abs(vector[i])<rtsdv)
   {
    zcount++;
   }
   else zcount = 0;
   zrun = (zcount>zrun) ? zcount : zrun;
  }

  printf("   high= %6.2f, low= %6.2f, size= %8i \n", high, low,size);
  printf("   mean = %4.1f, var = %8.2f, stddev = %6.2f\n", mean, var, stddev);
  printf("   zerothresh = %6.2f, zeros = %i, longest zero run = %i\n",
     rtsdv,zeros,zrun);
  printf("   outsiderthresh = %6.2f, outsiders = %i\n",sd4,outsiders);


  if (quantize==1) quant(vector,headvector,size,(int)stddev*100,
                  (int)mean,debug);
}


/*************************************************************************/
void main(int argc, char *argv[])
{
 int i, quantize, quant, zero;
 int xdim=0, ydim=0, size=0, level=0, sect=0, cursize=0;
 float *vector;
 float *headvector;
 char string[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}, *suf=".bq";

 if(argc < 5)
 {
  printf("Usage pgm <filename> xdim ydim Q Z {1/-1}  \n");
  printf("First name is source file\n");
```

```
  printf("Q=Levels to quantize              \n");
  printf("Z=Levels to zero  \n");
  printf("1 quantize, -1 unquantize, 0 none\n");
  exit(0);
}

infile = fopen(argv[1],"rb");
if(infile == NULL)
{
 printf("File open failure\n");
 exit(0);
}
strcpy(string,argv[1]);
strcat(string,".q");
outfile = fopen(string,"wb");
if(infile == NULL)
{
 printf("File open failure\n");
 exit(0);
}

xdim = atoi(argv[2]);
ydim = atoi(argv[3]);
quant = atoi(argv[4]);
zero  = atoi(argv[5]);
quantize = atoi(argv[6]);
level = quant+zero;

if (abs(quantize)>1) quantize = 1;
size = xdim*ydim;

vector = getvect(size,sizeof(float));
headvector=getvect(2*level,sizeof(float));

if (quantize==1) fread(vector,sizeof(float),size,infile);
else
{
 fread(&level,sizeof(int),1,infile);
 fread(headvector,sizeof(float),2*level,infile);
 fread(vector,sizeof(float),size,infile);
}

for (i=level;i>0;i--)
{
 cursize = size/(int)pow(2.0,(double)i);
 if (level==i)
 {
  printf("For level %i, sect %i \n", level,1);
  sectquant(vector,headvector,cursize,0,quantize,1);
 }
 if(i<=zero) for (sect=2;sect<=2;sect++)
 {
  printf("For level %i, sect %i zeroing\n",i,sect);
  sectquant(&vector[cursize],&headvector[i*2],
```

```
            cursize,1,quantize,1);
    }
    else for (sect=2;sect<=2;sect++)
    {
     printf("For level %i, sect %i quantizing\n",i,sect);
     sectquant(&vector[cursize],&headvector[i*2],
            cursize,0,quantize,1);
    }
  }


  if (quantize==1)
  {
   fwrite(&level,sizeof(int),1,outfile);
   fwrite(headvector,sizeof(float),level*2,outfile);
   fwrite(vector,sizeof(float),size,outfile);
   free(vector);
   free(headvector);
  }
  else
  {
   free(headvector);
   fwrite(vector,sizeof(float),size,outfile);
   free(vector);
  }
  fclose(infile);
  fclose(outfile);
}
```

```
/*-------------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------------

FILENAME: QONLY2.C

PURPOSE: Quantizes an input float vector from a wavelet
transform file

Input file is a binary float file
output file is a char file

Creates a data header of 1 + 2*levels processed (floats)
  Containing levels processed and mean and standard deviation of each
  level processed.

FEATURES: Uses a bin calculation routine that is based on the desired range of the bins
-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <string.h>

#define DEBUG 1
float *VECT, *HVECT;
FILE *infile, *outfile, *binfile;


/*****************************************************/
void * getvect(int n,int size)
{
  void *vectptr;

  vectptr = (void *) calloc(n,size);
  if(vectptr == NULL) {
    printf("Error: dynamic memory fail\n");
    exit(0);
  }
  return(vectptr);
}

/*****************************************************/
void intquant(float *vector,float *headvector,int size,
        int stddev,int mean,int debug,int numbins)
{
  float fstddev, fmean, sd3 ,binsize;
  int i=0;

  headvector[0] = (float)mean;
  headvector[1] = (float)stddev;

  fstddev = ((float)stddev)/100.0;
  fmean = ((float) mean)/100.0;
```

```
  /* put most in char range */
  sd3 = fstddev*3;
  binsize= sd3/numbins;

  while (i<size)
  {
   if (abs(vector[i]-fmean)<binsize)
   {
    vector[i] = 0;
   }
   else vector[i] = (vector[i]-fmean)/binsize;
   i++;
  }
}

/*************************************************/
void intunquant(float *vector,float *headvector,int size,int debug,int numbins)
{
 int i=0, stddev, mean=0;
 float fstddev, fmean, temp, sd3, binsize, halfbin;

 mean = (int)headvector[0];
 stddev = (int)headvector[1];

 fstddev = ((float)stddev)/100.0;
 fmean = ((float)mean)/100.0;
 sd3 = fstddev*3;
 binsize= sd3/numbins;
 halfbin = binsize/2;
 printf("    mean = %6.2f, stddev = %6.2f, 3sd = %6.2f, binsize = %6.2f\n",
      fmean,fstddev,sd3,binsize);

 while (i<size)
 {
  if(vector[i]!=0) vector[i]=(vector[i]*binsize)+fmean;
  else vector[i]=fmean;
  i++;
 }
}

/*************************************************/
void quant(float *vector,float *headvector,int size,
        int stddev,int mean,int debug,int numbins)
{
 float fstddev, fmean, frtsdv,sd3;
 int i=0;

 fstddev = ((float)stddev)/100.0;
 fmean = ((float) mean)/100.0;
 sd3 = fstddev*3;
 frtsdv = sd3/numbins;


 headvector[0] = (float)mean;
```

```
  headvector[1] = (float)stddev;

  while (i<size)
  {
   if (abs(vector[i]-fmean)<frtsdv)
   {
    vector[i] = 0;
   }
   else vector[i] = (vector[i]-fmean)/frtsdv;
   i++;
  }
}

/*************************************************/
void unquant(float *vector,float *headvector,int size,int debug,int numbins)
{
 int i=0, stddev, mean=0, rebuilt;
 float fstddev, fmean, temp;
 float sd3, binsize, halfbin;

 mean = (int)headvector[0];
 stddev = (int)headvector[1];
 fstddev = ((float)stddev)/100.0;
 fmean = ((float)mean)/100.0;
 sd3 = fstddev*3;
 binsize = sd3/numbins;
 halfbin = binsize/2;

 printf("mean = %6.2f, stddev = %6.2f, 3sd = %6.2f, binsize = %6.2f\n",
      fmean,fstddev,sd3,binsize);

 while (i<size)
 {
  if(vector[i]!=0) vector[i]=(vector[i]*binsize)+fmean;
  else vector[i]=fmean;
  i++;
 }
}

/*************************************************/
void sectquant(float *vector,float *headvector, int size,int zeroout,int quantize,int debug,int
           intqnt,int numbins)
{
 int i;
 double high=-10000000.0, low=10000000.0;
 int zeros=0, zrun=0, zcount=0, outsiders=0;
 double sum=0.0, squares=0.0, mean=0.0, var=0.0, stddev=0.0, sd3=0.0;
 double rtsdv=0.0;
 float binsize;

 if (quantize==-1)
 {
  if(intqnt) intunquant(vector,headvector,size,debug,numbins);
  else unquant(vector,headvector,size,debug,numbins);
```

```
  }

  for (i=0;i<size;i++)
  {
   high = ((double)vector[i]>high) ? (double)vector[i] : high;
   low  = ((double)vector[i]<low ) ? (double)vector[i] : low ;
   sum += (double)vector[i];
   squares += (double)vector[i]*(double)vector[i]/(double)size;
  }
  mean = sum/(double)size;
  var = squares - (mean*mean);
  stddev = sqrt(var);
  sd3 = 3*stddev;
  binsize = sd3/numbins;  /* zero threshold, 3*std dev /bins */

  for (i=0;i<size;i++)
  {
   if ((double)abs(vector[i]-mean)>sd3) outsiders++;
   if ((double)abs(vector[i]-mean)<binsize)
   {
    zeros++;
   }
   else if(zeroout)
   {
    zeros++;
    vector[i]=0.0;
   }
   if ((double)abs(vector[i]-mean)<binsize)
   {
    zcount++;
   }
   else zcount = 0;
   zrun = (zcount>zrun) ? zcount : zrun;
  }
  if (quantize==1) printf("    mean = %4.1f, var = %8.2f, stddev = %6.2f\n",
                    mean, var, stddev);
  printf("    high = %6.2f, low = %6.2f, size = %8i, bins = %i \n",
       high, low,size,numbins);
  printf("    binsize = %6.2f, zeros = %i, longest zero run = %i\n",
       binsize,zeros,zrun);
  printf("    outsiderthresh = %6.2f, outsiders = %i\n",sd3,outsiders);

  if (quantize==1)
  {
   if (intqnt) intquant(vector,headvector,size,(int)(stddev*100),
                    (int)(mean*100),debug,numbins);
   else quant(vector,headvector,size,(int)(stddev*100),
                    (int)(mean*100),debug,numbins);
  }
}

/*************************************************************************/
void main(int argc, char *argv[])
{
```

```
int i, quantize, quant, zero=0;
int xdim=0, ydim=0, size=0, level=0, sect=0, cursize=0;
int numbins;
char clevel;
float flevel;
float *vector;
float *headvector;
char string[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}, *suf=".bq";

if(argc < 4)
{
  printf("Usage pgm <filename> Q {1/-1}  \n");
  printf("First name is source file\n");
  printf("Q=Levels to quantize              \n");
  printf("1 quantize, -1 unquantize, 0 none\n");
  exit(0);
}
infile = fopen(argv[1],"rb");
if(infile == NULL)
{
 printf("File open failure\n");
 exit(0);
}
strcpy(string,argv[1]);
strcat(string,".q2");
outfile = fopen(string,"wb");
if(outfile == NULL)
{
 printf("File open failure\n");
 exit(0);
}
level = atoi(argv[2]);
quantize = atoi(argv[3]);
if (abs(quantize)>1) quantize = 1;
size = 512*512;
vector = (float *)getvect(size,sizeof(float));

if (quantize==1)
{
  fread(vector,sizeof(float),size,infile);
  headvector=(float *)getvect(2*(level+1),sizeof(float));
  flevel = (float)level;
}
else
{
  fread(&flevel,sizeof(float),1,infile);
  printf("%i levels to process\n",(int)flevel);
  level = (int)flevel;
  headvector=(float *)getvect(2*(level+1),sizeof(float));
  fread(headvector,sizeof(float),2*(level+1),infile);
  fread(vector,sizeof(float),size,infile);
}

for (i=level;i>0;i--)
```

```
 {
  cursize = size/(int)pow(2.0,(double)i);
  if (level==i)
  {
   numbins=100;
   printf("For level %i, sect %i int quantizing, %i bins \n", level,1,numbins);
   sectquant(vector,&headvector[(level-i)*2],cursize,0,quantize,1,1,numbins);
  }
  for (sect=2;sect<=2;sect++)
  {
   numbins=(int)(100.0/flevel*(float)i);
   printf("For level %i, sect %i quantizing, %i bins\n",i,sect,numbins);
   sectquant(&vector[cursize],&headvector[(level-i+1)*2],
         cursize,0,quantize,1,0,numbins);
  }
 }

 if (quantize==1)
 {
  fwrite(&flevel,sizeof(float),1,outfile);
  fwrite(headvector,sizeof(float),(level+1)*2,outfile);
  fwrite(vector,sizeof(float),size,outfile);
  free(vector);
  free(headvector);
 }
 else
 {
  free(headvector);
  fwrite(vector,sizeof(float),size,outfile);
  free(vector);
 }
 fclose(infile);
 fclose(outfile);
}
```

```
/*-------------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------------*/

FILENAME: FSTATS.C

PURPOSE: Calculates the distribution of an input char file

Input file is a char  file
output file is a text file

-------------------------------------------------------------
-------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINE 512
#define NR_END 1
#define FREE_ARG char*
#define MC 512
#define MQ (2*MC-1)



long filesize(FILE *stream)
{
  long curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}


int getfilename(char name[25],int argc,char *argv[])
{
 FILE *in;
 char destination[25],filename[25];
 char string[25];
 int i,namelength,error=1;

 while(error)
 {
  for (i=0;i<25;i++)
  {
   string[i]=NULL;
   filename[i]=NULL;
   name[i]=NULL;
  }
  if(argc==1)
  {
```

```
      printf("usage:  prgm filename ");
      exit(0);
    }
    else
    {
      strcpy(string,argv[1]);
      argc=1;
    }

    printf("opening %s\n",string);
    if ((in = fopen(string, "rb")) == NULL)
    {
      fprintf(stderr,"Cannot open %s file, press return \r",filename);

      error=getchar();

      fprintf(stderr,"                                    \r");
      error=1;
    }
    else
    {
      fclose(in);
      error=0;
    }
  }
  return 1;
}


/*************************************************************************
  MAIN   -  STATISTICS ACCUMULATION DRIVER
*************************************************************************/
int main(int argc,char *argv[])
{
  long nfreq[256];
  FILE *out, *in, *temp, *swap;
  FILE *tbl;
  char filename[25],
    name[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    ext[4]={0,0,0,0},
    headerbuf[200];
  unsigned int i,j, getsize;
  long fsize=0,runlen=0,count,charcount;
  unsigned long *table=NULL;
  double numbits;
  int index;
  unsigned char inchar, last;
  int compress=0;
  float entropy=0.0,prob=0.0,inf_num=0.0,inf_sum=0.0,avg_len=0.0, sigma2=0.0;
  float log2=log(2);
  float mean=0.0, meansq=0.0;
  double sqsum=0.0,maxenergy=0.0;
```

```
for(i=0;i<25;i++) filename[i]=NULL;
strcpy(filename,argv[1]);

printf("opening in  %s\n",filename);

if ((in = fopen(filename,"rb")) == NULL)
{
 fprintf(stderr,"Input %s not found.\n",filename);
 exit(1);
}

strncat(filename,".sts",4);

printf("opening out %s\n",filename);

if ((tbl = fopen(filename,"w")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",filename);
 exit(1);
}

printf("Analyzing file and building statistics\n");

fsize=filesize(in);

for (j=0; j<256; j++)
{
 nfreq[j]=0;
}

count=0;
charcount=0;
last=255; /* put last out of range */

for(;;) /* character counts*/
{
 inchar=(unsigned char)fgetc(in);
 index=((int)inchar); /* +128;*/
 if(feof(in)) break;
 nfreq[index]++;
 count++;
}


fprintf(tbl,"ind    nfreq    prob    inf num \n\n");
printf("ind    nfreq    prob    inf num \n\n");
for (j=0; j<256;j++)
{
 if (nfreq[j])
 {
   prob= (float)nfreq[j]/count;
   inf_num= -(log(prob)/log2);
   inf_sum+= inf_num;
```

```
    sqsum += j*j*nfreq[j];
    entropy+= prob*inf_num;
    mean+=(float)j*prob;
    printf("%3i,  %6i,  %6.4f,  %6.4f \n",j,nfreq[j],prob,inf_num);
    fprintf(tbl,"%3i,  %6i,  %6.4f,  %6.4f \n",j,nfreq[j],prob,inf_num);
   }
  }

  meansq = mean*mean;

  for (j=0; j<256;j++)
  {
   sigma2+= (float)nfreq[j]*(((float)j-mean)*((float)j-mean));
  }

  sigma2 = sigma2/(float)(count);

  fprintf(tbl,"\n\nFILE_STATISTICS\n");
  fprintf(tbl,"total_char= %lu\n",count);
  fprintf(tbl,"info_number= %f\n",inf_sum);
  fprintf(tbl,"entropy= %f\n",entropy);
  fprintf(tbl,"mean= %f\n", mean);
  fprintf(tbl,"variance= %f\n",sigma2);
  fprintf(tbl,"std_dev= %f\n",sqrt((double)sigma2));
  fprintf(tbl,"total_energy= %f\n",sqsum);

  printf("ENCODING STATISTICS\n");
  printf("total char read      = %lu\n",count);
  printf("information number    = %f\n",inf_sum);
  printf("file entropy         = %f\n",entropy);
  printf("mean                 = %f\n", mean);
  printf("variance             = %f\n",sigma2);
  printf("std deviation         = %f\n",sqrt((double)sigma2));
  printf("total_energy         = %f\n",sqsum);

  fclose(tbl);
  fclose(in);
  return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: FCMP.C

PURPOSE:  Builds a comparison error distribution

Input files are char files
output file is a text file

-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINE 512
#define NR_END 1
#define FREE_ARG char*
#define MC 512
#define MQ (2*MC-1)


long filesize(FILE *stream)
{
  long curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}


/*****************************************************************************

 MAIN   - STATISTICS ACCUMULATION DRIVER

*****************************************************************************/

int main(int argc,char *argv[])
{
 long nfreq[256];
 FILE *out, *in,*in2, *temp, *swap;
 FILE *tbl;
 char filename[25],
   name[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
   ext[4]={0,0,0,0},
   headerbuf[200];
 unsigned int i,j, getsize;
 long fsize=0,runlen=0,count,charcount;
 unsigned long *table=NULL;
```

```
double numbits;
int index;
unsigned char inchar, inchar2, last;
int compress=0;
float entropy=0.0,prob=0.0,inf_num=0.0,inf_sum=0.0,avg_len=0.0, sigma2=0.0;
float log2=log(2);
float mean=0.0, meansq=0.0;
double sqsum=0.0,maxenergy=0.0;

for(i=0;i<25;i++) filename[i]=NULL;
strcpy(filename,argv[1]);

printf("opening in  %s\n",filename);

if ((in = fopen(filename,"rb")) == NULL)
{
 fprintf(stderr,"Input %s not found.\n",&argv[1][0]);
 exit(1);
}

if ((in2 = fopen(argv[2],"rb")) == NULL)
{
 fprintf(stderr,"Input %s not found.\n",&argv[2][0]);
 exit(1);
}

strncat(filename,".dif",4);

printf("opening out %s\n",filename);

if ((tbl = fopen(filename,"w")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",filename);
 exit(1);
}

printf("Analyzing file and building statistics\n");

fsize=filesize(in);

for (j=0; j<256; j++)
{
 nfreq[j]=0;
}

count=0;
charcount=0;
last=255; /* put last out of range */

for(;;) /* character counts*/
{
 inchar=(unsigned char)fgetc(in);
 inchar2=(unsigned char)fgetc(in2);
 index=((int)inchar-inchar2+128); /* +128;*/
```

```
  if(feof(in)) break;
  nfreq[index]++;
  count++;
 }

 fprintf(tbl,"ind   nfreq   prob   inf num \n\n");
 printf("ind   nfreq   prob   inf num \n\n");
 for (j=0; j<256;j++)
 {
  if (nfreq[j])
  {
   prob= (float)nfreq[j]/count;
   inf_num= -(log(prob)/log2);
   inf_sum+= inf_num;
   sqsum += (j-128)*(j-128)*nfreq[j];
   entropy+= prob*inf_num;
   mean+=(float)(j)*prob;
   printf("%3i,  %6i,  %6.4f,  %6.4f \n",j-128,nfreq[j],prob,inf_num);
   fprintf(tbl,"%3i,  %6i,  %6.4f,  %6.4f \n",j-128,nfreq[j],prob,inf_num);
  }
 }
 meansq = mean*mean;
 for (j=0; j<256;j++)
 {
  sigma2+= (float)nfreq[j]*(((float)j-mean)*((float)j-mean));
 }
 sigma2 = sigma2/(float)(count);

 fprintf(tbl,"\n\nFILE_COMPARISON_STATISTICS\n");
 fprintf(tbl,"%s - %s\n\n",&argv[1][0],&argv[2][0]);
 fprintf(tbl,"total_char= %lu\n",count);
 fprintf(tbl,"info_number= %f\n",inf_sum);
 fprintf(tbl,"entropy= %f\n",entropy);
 fprintf(tbl,"mean= %f\n", mean-128.0);
 fprintf(tbl,"variance= %f\n",sigma2);
 fprintf(tbl,"std_dev= %f\n",sqrt((double)sigma2));
 fprintf(tbl,"total_energy= %f\n",sqsum);

 printf("DIFFERENCE STATISTICS\n");
 printf("%s - %s\n\n",argv[1],argv[2]);
 printf("total char read      = %lu\n",count);
 printf("information number   = %f\n",inf_sum);
 printf("file entropy         = %f\n",entropy);
 printf("mean                 = %f\n", mean-128.0);
 printf("variance             = %f\n",sigma2);
 printf("std deviation        = %f\n",sqrt((double)sigma2));
 printf("total_energy         = %f\n",sqsum);

 fclose(tbl);
 fclose(in);
 fclose(in2);
 return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: RGBLOG.C

PURPOSE: Splits a 24-bit RGB file into three planes using a
transform based on the human visual system

Input file is a 24-bit color file
3 char output files created containing a plane each

NOTE: similar to RGBKKK and KKKRGB see notes for inverse transform
-------------------------------------------------------
-------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

/* file size determinator */
long filesize(FILE *stream)
{
  long curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}


/*******************MAIN*************************************/

int main(int argc,char *argv[])
{
 FILE *in, *R, *G, *B, *Y, *I, *Q, *YT, *IT, *QT;
 int i,j,bmp=0;
 char filename[12],othername[12]={0,0,0,0,0,0,0,0,0,0,0,0},
   name[8]={0,0,0,0,0,0,0,0},
   ext[4]={0,0,0,0},
   headerbuf[100];
 unsigned char RGB[3],KKK2[3];
 float KKK[3];
 int width,height;
 unsigned long count=0;
 long size;
 int textout=0;

 if (argc<2) printf("usage: pgm filename [t] {for text output}\n");

 strcpy(name,argv[1]);

 if ((in = fopen(argv[1],"rb")) == NULL)
```

```
{
 fprintf(stderr,"Input %s not found.\n",filename);
 exit(1);
}

size=filesize(in);
height=width=(int)sqrt((double)size/3);
if((long)height*width*3-size!=0)
{
 printf("filesize error\n");
 exit(1);
}

printf("height=%i, width=%i, size=%lu\n",height,width,(unsigned long)height*width);

for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".l1",4);

if ((Y = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
}

for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".l2",4);

if ((I = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
}

for (i=0;i<=14;i++) othername[i]=0;
strcpy(othername,name);
strncat(othername,".l3",4);

if ((Q = fopen(othername,"wb")) == NULL)
{
 fprintf(stderr,"Output file %s not created.\n",othername);
 exit(1);
}

for(i=0;i<height*width;i++)
{
 fread(RGB,1,3,in);

 KKK[0]=(0.299*(float)RGB[0]+0.587*(float)RGB[1]+0.114*(float)RGB[2]);
 KKK[1]=(0.607*(float)RGB[0]+0.174*(float)RGB[1]+0.201*(float)RGB[2]);
 KKK[2]=(                0.066*(float)RGB[1]+1.117*(float)RGB[2]);

 KKK[0]=log(KKK[0]);
```

```
    KKK[1]=log(KKK[1]);
    KKK[2]=log(KKK[2]);

    KKK2[0]=(unsigned char)((21.50*KKK[0]+106.277)/.6197);
    KKK2[1]=(unsigned char)((-41.0*KKK[0]+41.00*KKK[1]+21.652)/.1344);
    KKK2[2]=(unsigned char)((-6.27*KKK[0]+6.27*KKK[2]+17.771)/.0941);


    fwrite(&KKK2[0],1,1,Y);
    fwrite(&KKK2[1],1,1,I);
    fwrite(&KKK2[2],1,1,Q);
  }

  printf("\n");

  fclose(in);
  fclose(Y);
  fclose(I);
  fclose(Q);
  return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: VF.C

PURPOSE: Float file viewer for the SGI machines, displays
values from 0 - 255 truncates values beyond that to 0 or 255

Input file is a binary float file

NOTE: 8-bit greyscale filesare similarly handled just make the
appropriate changes to handle and display char based files
-------------------------------------------------------
-------------------------------------------------------*/
#include <gl/gl.h>
#include <gl/device.h>
#include <stdio.h>
#include <fcntl.h>
#include <math.h>

/*****************************************************/
void * getvect(int n,int siz)
{
  void *ptr;

  ptr = (void *) calloc(n,siz);
  if(ptr == NULL) {
    printf("Error: Dynamic memory allocation fail\n");
    exit(0);
  }
  return(ptr);
}

/*****************************************************/
/* file size determinator */
int  filesize(FILE *stream)
{
   int  curpos, length;
   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}


/*****************************************************/
main(int argc,char *argv[])
{
  int x;
  int y;
  FILE *fp;
  int i;
```

```
int n;
int X=0;
int Y=0;
int C;
int waiting =1;
int notdone =1;
Device dev;
int size,hsize=0;
float scaler=1.0;
int offset=0;
short colo[256][3];
long vert[2];
float *c;
float *image, *header;
short val;

if(argc < 2) {
  printf("This program is for float format files \n");
  printf("Usage: pgm filename [options]\n");
  printf("-s# scaler = multipler for values float#\n");
  printf("-o# offset = offset values by int#\n");
  printf("-h# size of header          \n");
  printf("-x# force width to int#\n");
  printf("-y# force width to int#\n");
  printf("-b windows bmp format {not implemented}\n");
  printf("if the file is one of the standard sizes it will automatically determine the size\n");
  printf("in program: up & down arrow keys scale *2|*.5\n");
  printf("           left & right arrow keys offset -20|+20\n");
  exit(0);
}

fp=fopen(argv[1],"rb");
if (fp==NULL)
{
   printf("file open error\n");
   exit(0);
}

for (i=2;i<argc;i++)
{
  switch (argv[i][1])
  {
     case 'x':X = atoi(&argv[i][2]);
              break;
     case 'y':Y = atoi(&argv[i][2]);
              break;
     case 'o':offset = atoi(&argv[i][2]);
              break;
     case 's':scaler = atof(&argv[i][2]);
              break;
     case 'h':hsize = atoi(&argv[i][2]);
              header = (float *) getvect(hsize ,sizeof(float));
              fread(header,sizeof(float),hsize,fp);
              break;
```

```
  }
}

size = filesize(fp)/sizeof(float);

switch (size-hsize)
{
  case 262144:X=512;
              Y=512;
              break;
  case 65536 :X=256;
              Y=256;
              break;
  case 64000 :X=320;
              Y=200;
              break;
  case 16386 :X=128;
              Y=128;
              break;
  case 4096  :X=64 ;
              Y=64 ;
              break;
  default    :if(X>0&&Y==0)
                Y=(size-hsize)/X;
              else if(Y>0&&X==0)
                X=(size-hsize)/Y;
              else if(Y+X==0)
                X=Y=(int)sqrt((double)size-hsize);
}

image = (float *) getvect(X*Y,sizeof(float));
prefsize(X,Y);
winopen(argv[1]);
ortho2(0.0,X-1,-Y+1,0.0);
cmode();
gconfig();
color(WHITE);
clear();
qdevice(ESCKEY);
qdevice(REDRAW);
qdevice(RIGHTARROWKEY);

qdevice(LEFTARROWKEY);
qdevice(UPARROWKEY);
qdevice(DOWNARROWKEY);

n=fread(image,sizeof(float),X*Y,fp);

for(x=1;x<256;x++) {
  mapcolor(x,x,x,x);
}
gconfig();
while(notdone)
{
```

```
c=image;
for(y=0;y<Y;y++) {
 for(x=0;x<X;x++) {
   if  ((int)(*c+offset)*scaler>255) C = 255;
   else if ((int)(*c+offset)*scaler< 0) C = 0;
   else C = (int)(*c+offset)*scaler;
   color(C);
   c++;
   bgnpoint();
   vert[0]=x;
   vert[1]=-y;
   v2i(vert);
   endpoint();
   gflush();
 }
}

waiting=1;
while(waiting)
{
  while (! qtest()){}
  switch(dev=qread(&val))
  {
     case REDRAW: waiting=0;
          break;
     case UPARROWKEY:
     case DOWNARROWKEY:
     case RIGHTARROWKEY:
     case LEFTARROWKEY:
     case ESCKEY: if(val==0) waiting=0;
          break;
  }
}
switch(dev)
{
 case ESCKEY : notdone = 0; break;
 case UPARROWKEY:
          scaler*=2;
          printf ("scaler set to %5.2f\n",scaler);
          break;
 case DOWNARROWKEY:
          scaler/=2;
          printf ("scaler set to %5.2f\n",scaler);
          break;
 case RIGHTARROWKEY: offset += 20;
          printf ("offset set to %3i\n",offset);
          break;
 case LEFTARROWKEY:  offset -= 20;
          printf ("offset set to %3i\n",offset);
          break;
 case REDRAW :

}
/*  while(!(qtest() && qread(&val) == ESCKEY && val == 0)) { }*/
```

```
  }
  free(image);
  free(header);
  gconfig();
  gexit();
  return 0;
}


/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: VCOLOR.C

PURPOSE: Displays 24-bit rgb images on the SGI machines

Input file is a 24-bit rgb file also will handle windows BMP files
-------------------------------------------------------
-------------------------------------------------------*/
#include <gl/gl.h>
#include <gl/device.h>
#include <stdio.h>
#include <fcntl.h>
#include <math.h>

/*****************************************************/
void * getvect(int n,int siz)
{
  void *ptr;
  ptr = (void *) calloc(n,siz);
  if(ptr == NULL) {
    printf("Error: Dynamic memory allocation fail\n");
    exit(0);
  }
  return(ptr);
}

/*****************************************************/
/* file size determinator */
int  filesize(FILE *stream)
{
   int  curpos, length;
   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}


/*****************************************************/
main(int argc,char *argv[])
{
```

```
int x;
int y;
FILE *fp;
int i,j;
int n;
int X=0;
int Y=0;
float C;
int bmp=0;
int waiting =1;
int notdone =1;
Device dev;
int size,hsize=0;
float scaler=1.0;
float offset=0;
short colo[256][3];
long vert[2];
float blackvect[3] = {0.0,0.0,0.0};
float colovect[3];
unsigned char *c;
unsigned char *image, *header;
short val;

if(argc < 2) {
  printf("This program is for unsigned char format files \n");
  printf("Usage: pgm filename [options]\n");
  printf("-s# scaler = multipler for values float#\n");
  printf("-o# offset = offset values by float#\n");
  printf("-h# size of header          \n");
  printf("-x# force width to int#\n");
  printf("-y# force width to int#\n");
  printf("-b windows bmp format \n");
  printf("if file is one of the standard sizes it will automatically determine the size\n");
  printf("in program: up & down arrow keys scale *2|*.5\n");
  printf("          left & right arrow keys offset -.05|+.05\n");
  exit(0);
}

fp=fopen(argv[1],"rb");
if (fp==NULL)
{
   printf("file open error\n");
   exit(0);
}

size = filesize(fp)/(sizeof(unsigned char));

for (i=2;i<argc;i++)
{
  switch (argv[i][1])
  {
     case 'x':X = atoi(&argv[i][2]);
              break;
     case 'y':Y = atoi(&argv[i][2]);
```

```
            break;
    case 'o':offset = atof(&argv[i][2]);
            break;
    case 's':scaler = atof(&argv[i][2]);
            break;
    case 'h':hsize = atoi(&argv[i][2]);
            header = (unsigned char *) getvect(hsize ,sizeof(unsigned char));
            fread(header,sizeof(unsigned char),hsize,fp);
            break;
    case 'b':hsize = 54;
            header = (unsigned char *) getvect(hsize ,sizeof(unsigned char));
            bmp = 1;
            fread(header,sizeof(unsigned char),hsize,fp);
            break;
  }
}

size -= hsize;
size /= 3;

switch (size)
{
  case 262144:X=512;
            Y=512;
            break;
  case 65536 :X=256;
            Y=256;
            break;
  case 64000 :X=320;
            Y=200;
            break;
  case 16386 :X=128;
            Y=128;
            break;
  case 4096  :X=64 ;
            Y=64 ;
            break;
  default    :if(X>0&&Y==0)
              Y=(size)/X;
            else if(Y>0&&X==0)
              X=(size)/Y;
            else if(Y+X==0)
              X=Y=(int)sqrt((double)size);
}

printf("image is %i bytes with a %i header, X=%i by Y=%i\n",size,hsize,X,Y);
image = (unsigned char *) getvect(size*3,sizeof(unsigned char));
prefsize(X,Y);
winopen(argv[1]);
ortho2(0.0,X-1,-Y+1,0.0);
RGBmode();
gconfig();
color(WHITE);
c3f(blackvect);
```

```
clear();
qdevice(ESCKEY);
qdevice(REDRAW);
qdevice(RIGHTARROWKEY);
qdevice(LEFTARROWKEY);
qdevice(UPARROWKEY);
qdevice(DOWNARROWKEY);
n=fread(image,sizeof(unsigned char),size*3,fp);
gconfig();
while(notdone)
{
 c=image;
 for(y=0;y<Y;y++)
 {
  for(x=0;x<X;x++)
  {
   if (bmp)
   {
    for (j=2;j>=0;j--)
    {
     if  ((((float)*c/255.0)*scaler)+offset>=1.0) C = 0.99;
     else if ((((float)*c/255.0)*scaler)+offset< 0.0) C = 0.0;
     else C = (((float)*c/255.0)*scaler)+offset;
     colovect[j]=C;
     c++;
    }
    c3f(colovect);
    bgnpoint();
    vert[0]=X-x;
    vert[1]=-Y+y;
   }
   else
   {
    for (j=0;j<3;j++)
    {
     if  ((((float)*c/255.0)*scaler)+offset>=1.0) C = 0.99;
     else if ((((float)*c/255.0)*scaler)+offset< 0.0) C = 0.0;
     else C = (((float)*c/255.0)*scaler)+offset;
     colovect[j]=C;
     c++;
    }
    c3f(colovect);
    bgnpoint();
    vert[0]=x;
    vert[1]=-y;
   }

   v2i(vert);
   endpoint();
   gflush();
  }
}

waiting=1;
```

```
    while(waiting)
    {
      while (! qtest()){}
      switch(dev=qread(&val))
      {
         case REDRAW: waiting=0;
                  break;
         case UPARROWKEY:
         case DOWNARROWKEY:
         case RIGHTARROWKEY:
         case LEFTARROWKEY:
         case ESCKEY: if(val==0) waiting=0;
                  break;
      }
    }
    switch(dev)
    {
     case ESCKEY : notdone = 0; break;
     case UPARROWKEY:
              scaler*=2.0;
              printf ("scaler set to %5.2f\n",scaler);
              break;
     case DOWNARROWKEY:
              scaler/=2.0;
              printf ("scaler set to %5.2f\n",scaler);
              break;
     case RIGHTARROWKEY: offset += .05;
              printf ("offset set to %3.0f\n",offset);
              break;
     case LEFTARROWKEY:  offset -= .05;
              printf ("offset set to %3.0f\n",offset);
              break;
     case REDRAW :

    }
/*   while(!(qtest() && qread(&val) == ESCKEY && val == 0)) { }*/
  }
  free(image);
  free(header);
  gconfig();
  gexit();
  return 0;
}
```

```
/*-------------------------------------------------------
THOMAS W. PIKE, Thesis Project, Spring '95
-------------------------------------------------------

FILENAME: VIEWRGB.C

PURPOSE: PC based color image viewer
-------------------------------------------------------
-------------------------------------------------------*/
/*************************************************************************
*  ----------------------------------------------          *
*  THOMAS W. PIKE, SPRING 95, VERS 2.0                      *
*  ----------------------------------------------          *
*                                                          *
*  PURPOSE:  RGBcolor graphics viewer for SVGA systems      *
*                                                          *
*  FEATURES: automatic file size detection routines, builds a 256 psuedo
*  color map for 24 bit images                             *
*                                                          *
*. NOTES: compiled with turbo C++ source option on         *
*                                                          *
*************************************************************************/
#include <stdlib.h>
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <svga256.h>
#include <string.h>

#define MAX_WIDTH 2048

/************************************************************************/
/*                                                          */
/* FUNCTION FILESIZE returns the filesize                   */
/*                                                          */
/*                                                          */
/************************************************************************/
long filesize(FILE *stream)
{
  long curpos, length;
  curpos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, curpos, SEEK_SET);
  return length;
}


/************************************************************************/
/*                                                          */
/* PROCEDURE GETFILENAME                                    */
/*                                                          */
/*                                                          */
```

```
/*************************************************************************/
int getfilename(char *name, char *ext, int argc,char *argv[])
{
 FILE *in;
 char destination[25],filename[12];
 char string[20], *dot;
 int i,namelength,error=1;

 while(error)
 {
 for (i=0;i<20;i++)
 {
  string[i]=NULL;
  if(i<12) filename[i]=NULL;
  if(i<8)name[i]=NULL;
  if(i<4) ext[i]=NULL;
 }
 if(argc==1) {
  printf("viewrgb filename [-f -r]");
  printf("option f flips the order of display");
  printf("option r flips the rgb order to bgr");

 }
 else
 {
  strcpy(string,argv[1]);
  argc=1;
 }

 if ((in = fopen(string, "rb")) == NULL)
 {
  fprintf(stderr,"Cannot open %s file, press return \r",filename);

  error=getchar();

  fprintf(stderr,"                                    \r");
  error=1;
 }
 else
 {
  fclose(in);
  error=0;
 }
 }

 dot=strchr(string,'.');
 namelength = dot-string;
 strncpy(name, string, namelength);
 for(i=namelength;i<8;i++) name[i]=NULL;
 strncpy(ext, dot, 4);

 return 1;

}
```

```
/*************************************************************************/
/*                                                                       */
/* DetectVGA256 is a dummy function for the TURBO C initialization       */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
int huge DetectVGA256(void)
{
 return  0;
}


/*************************************************************************/
/*                                                                       */
/* FUNCTION setsvga256 graphics initialization                           */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
void setsvga256(void)
/*-----------------------------------------------------------------------
-----------------------------------------------------------------------
 GRAPHICS INITIALIZATION BLOCK
-----------------------------------------------------------------------
-----------------------------------------------------------------------*/
{
 /* select the svga256 driver and mode that supports the use */
 /* of the setrgbpalette function.                           */
 int gdriver, gmode, errorcode;
 int ht, xmax;

 /* Install the new driver */
 installuserdriver("SVGA256",DetectVGA256);
 gdriver=DETECT;

 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");

 /* read result of initialization */
 errorcode = graphresult();

 if (errorcode != grOk)
 /* an error occurred */
 {
  printf("Graphics error: %s\n", grapherrormsg(errorcode));
  printf("Press any key to halt:");
  getch();
  exit(1); /* terminate with an error code */
 }

 /*-----------------------------------------------------------------------
 -----------------------------------------------------------------------
 END OF GRAPHICS INITIALIZATION BLOCK
```

```
  ---------------------------------------------------------------------
  ---------------------------------------------------------------------*/
}

/*************************************************************************/
/*                                                                       */
/* FUNCTION SETColorSCALE creates 256 psuedo colorscale                  */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
void setcolorscale(void)
{
 int rcolor,gcolor,bcolor,ncolor;
 int i=0,j=0,k;

 /* create color scale */
 for (rcolor=0; rcolor<8; rcolor++)
 {
  for (gcolor=0; gcolor<8; gcolor++)
  {
   for (bcolor=0; bcolor<8; bcolor++)
   {

    ncolor=((rcolor<<6)+(gcolor<<3)+bcolor);
    if(ncolor%2==0)
    {
     ncolor/=2;
     printf("%7i\r",ncolor);
     setrgbpalette(ncolor,rcolor*8,gcolor*8,bcolor*8);
    }
   }
  }
 }

 /* display palette */
 for(i=0;i<256;i++)
 {
  for(j=0;j<8;j++)
  {
   putpixel(i*2,520+j,i);
   putpixel(i*2+1,520+j,i);
  }
 }
}

/*************************************************************************/
/*************************************************************************/
void main(int argc, char *argv[])
{
 /* initialization */
 long fsize;
 int i,j,k,l, ask=1;
 FILE * infile;
 int A,B,C, color;
```

```
char buffer[MAX_WIDTH], inchar;
unsigned char COLOR, OFFSET;
char filename[12],name[8],ext[4];
int header, width, height, offset;
float scaler;
int flip=0, reverse=0;

/* open file */
if(!getfilename(name,ext,argc,argv))
{
 fprintf(stderr,"Input file not found, terminating program\r");
 exit(0);
}

if (argc>2)
{
 for(i=2;i<argc;i++)
 {
  if(argv[i][0]='-')
  {
   switch(argv[i][1])
   {
    case 'f': flip=1;
              break;
    case 'r': reverse=1;
              break;
   }
  }
 }
}

for (i=0;i<12;i++) filename[i]=NULL;
for (i=0;i<8;i++) filename[i]=name[i];
strncat(filename,ext,4);
if ((infile = fopen(filename,"rb")) == NULL)
{
 fprintf(stderr,"Input %s not found.\n",filename);
 exit(1);
}
fsize=filesize(infile)/3;
/* calculate file parameters */
if(fsize>=64000&&fsize<66048)
{
 width=320;
 height=200;
 header=fsize-64000;
 if (header==0) ask=0;
}
else if(fsize>=128000&&fsize<128200)
{
 width=640;
 height=200;
 header=fsize-128000;
 if (header==0) ask=0;
```

```
}
else if(fsize>=224000&&fsize<224200)
{
 width=640;
 height=350;
 header=fsize-224000;
 if (header==0) ask=0;
}
else if(fsize>=245760&&fsize<245960)
{
 width=512;
 height=480;
 header=fsize-245760;
 if (header==0) ask=0;
}
else if(fsize>=307200&&fsize<307400)
{
 width=800;
 height=600;
 header=fsize-307200;
 if (header==0) ask=0;
}
else if(fsize>=745560&&fsize<745760)
{
 width=(int)(sqrt((double)fsize));
 height=width;
 header=fsize-width*height;
}
else
{
 width=(int)(sqrt((double)fsize));
 height=width;
 header=fsize-(width*height);
 header*=3;
}

if (ask)
{
 printf("Header size  = %i\n",header);
 printf("Image width  = %i\n",width);
 printf("Image height = %i\n",height);
 printf("Are these correct? [y,n]");
 scanf("%c",&inchar);
 printf("\n");
}

if (inchar=='n')
{
 printf("Image width  = ");
 scanf("%i",&width);
 printf("Image height = ");
 scanf("%i",&height);
 header=fsize-height*width;
 header*=3;
```

```
  printf("Header estimate  = %i\n",header);
  printf("Header size = ");
  scanf("%i",&header);
 }
 if (argc >= 3) offset= (int) atol(argv[2]);
 else offset=0;
 if (argc >= 4) scaler= atof(argv[3]);
 else scaler=1.0;
 /*-------------------------------------------------------------------
 -------------------------------------------------------------------
 GRAPHICS INITIALIZATION BLOCK
 -------------------------------------------------------------------
 -------------------------------------------------------------------*/
 setsvga256();
 setgraphmode(3);  /* mode 3 is 800X600 */
 setcolorscale();
/*-------------------------------------------------------------------
 -------------------------------------------------------------------
 END OF GRAPHICS INITIALIZATION BLOCK
 -------------------------------------------------------------------
 -------------------------------------------------------------------*/
 /* file display routine */
 fread(&buffer, sizeof(char), header, infile);

 for (i= 0; i<height; ++i)
 {
  if (fread(&buffer, sizeof(char), width*3, infile)==width*3)
  {
   for (j=0; j<width*3; j+=3)
   {
    /* scaler multiplies range by value to spread out or contract range
       offset changes the starting point of the color range, anything
       pushed above 255 with the shift will be left at 255
    */
    if(reverse) COLOR =(((buffer[j+2]/32)<<6)+((buffer[j+1]/32)<<3)+(buffer[j]/32))/2;
    else COLOR =(((buffer[j]/32)<<6)+((buffer[j+1]/32)<<3)+(buffer[j+2]/32))/2;
    if(flip) putpixel(width-j/3,height-i,(COLOR));
    else putpixel(j/3,i,(COLOR));
   }
  }
  else printf("full buffer not read");
 }
 /* wait until done viewing */
 getch();
 /* cleanup and exit */
 fclose(infile);
 closegraph();
 restorecrtmode();
}
```

Bibliography

[1] Bell, T. C., Cleary J. G., and Witten, I. H., *Text Compression*, Prentice Hall, 1990.

[2] Chui, C. K., *Wavelets: A Tutorial in Theory and Applications,* Academic Press, Inc., 1992

[3] Clarke, R. J., *Transform Coding of Images*, Microelectronics and Signal Processing Series, Academic Press, 1985.

[4] Cleary, J. G., and Witten, I. H., *Data Compression Using Adaptive Codiing and Partial String Matching*, IEEE Trans. on Communicatoin, Vol. Com-32, No. 4, pp. 396-402, Apr 1984.

[5] Coifman, R., Meyer, Y., and Wickerhauser, V.,Wavelet analysis and signal processing. In: *Wavelets and Their Applications,* Edited by Mary Beth Ruskai, Jones and Bartlet Publishers, 1991.

[6] Daubechies, I., *Ten Lectures on Wavelets,* Society for Industrial and Applied Mathematics, 1992.

[7] Daubechies, I., *Orthonormal bases of compactly supported wavelets*, Commun. Pure Appl. Math., 41 (7), 909-996, 1988.

[8] DeVore, R. and Lucier, B.J., *Wavelets*, Acta Numerica 11-56, 1991.

[9] Donoho, D., Johnstone, I., Kerkyacharian, G, and Picard, D., *Density estimation by wavelet thresholding*. Technical Report, Department of Statistics, Stanford University, 1993.

[10] Gersho, A., *Quantization*, IEEE Communications, pp. 16-29, Sep 1977.

[11] Gopinath, R. A. and Burrus, C. S., *Wavelets and Filter Banks*, Wavelets: A Tutorial in Theory and Applications, Edited by Chui, C. K., Academic Press, 1991

[12] Habibi, A., *Survey of Adaptive Image Coding Techniques*, IEEE Transactions on Communications, Vol. Com-25(11), pp. 1275-1284, Nov 1977.

[13] Habibi, A., *Comparison of nth-Order DPCM Encoder with Linear Transformations and Block Quantization Techniques*, IEEE Transactions on Communications Technology, Vol. COM-19(6), pp. 948-956, Dec 1971.

[14] Howard, P. G., and Vitter, J. S., *New methods for Lossless Image Compression Using Arithmetic Coding*, Proceedings of the Data Compression Conference, IEEE Computer Society Press, pp. 257-266, Apr 1991.

[15] Huffman, D. A., *A Method of the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE, Vol. 40(10), pp. 1098-1101, Sept 1952.

[16] Ifeachor, E. C., and Jervis, B. W., *Digital Signal Processing*, Addison-Wesley, 1993.

[17] Jayant, N. S., and Noll P., *Digital Coding of Waveforms*, Prentice Hall, 1984.

[18] Kunt, M., Bénard, M., and Leonardi, R., *Recent Results in High-Compression Image Coding*, IEEE Transactions on Circuits and Systems, Vol. CAS-34(11), pp 1306-1336, Nov 1987.

[19] Kunt, M., Ikonomopoulos, A., and Kocher, M., *Second Generation Image-Coding Techniques*, Proceedings of the IEEE, Vol. 73, No.4, pp 549-574, Apr 1985.

[20] Mallat, S.G., *A theory for multiresolution signal decomposition: the wavelet representation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 11 (7), 674-693, 1989.

[21] Nason, G.P. and Silverman B.W., *The discrete wavelet transform in S*, Statistics Research Report 93:07, University of Bath, Bath, BA27AY, UK, 1993.

[22] Nelson, M., *The Data Compression Book*, M&T Books, 1992.

[23] Pike, T. W. and Yfantis, E. A., *A Biased Pulse Code Modulation Algorithm for Image Compression*, Third Golden West Conference on Intelligent Systems, University of Nevada, Las Vegas, Jun 1994.

[24] Pratt, W. K., *Spatial Transform Coding of Colour Images*, IEEE Trans. on Communications, COM-19, pp. 980-992, 1971.

[25] Press W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes in C., Second Edition*, Cambridge University Press, 1993.

[26] Rabbani, M., *Selected Papers on Image Coding and Compression*, SPIE Milestone Series, Vol. MS48, 1991.

[27] Rabbani, M. and Jones, P. W., *Digital Image Compression Techniques"*, Vol. TT 7, SPIE Optical Engineering Press, 1991.

[28] Ruskai, M. B., *Wavelets and Their Applications,* Jones and Bartlett Publishers, 1991.

[29] Shannon, C. E., *A Mathematical Theory of Communication*, Bell Systems Technical Journal, Vol. 27, pp. 398-403, Jul 1948.

[30] Shapiro, J. M., *Embedded Image Coding Using Zerotrees of Wavelet Coefficients*, IEEE Trans. on Signal Processing, Vol. 41, No. 12, pp. 3445-3462, Dec 1993.

[31] Simoncelli, E. P, *Orthogonal Sub-band Image Transforms*, Masters Thesis, M. I. T., 1988.

[32] Wallace, G. K., *The JPEG Still Picture Compression Standard*, Communications of the ACM, Vol. 34, No. 4, pp. 31-44, Apr 1992

[33] Wei-Wei, L., and Gough, M. P., *A Fast-Adaptive Huffman Coding Algorithm*, IEEE Trans. on Communications, Vol. 41, No 4, pp. 535-538, Apr 1993.

[34] Witten, I. H., Radford, M. N., and Cleary, J. G., *Arithmetic Coding for Data Compression*, Communications of the ACM, Vol. 30(6), pp. 520-540, Jun 1987.

[35] Yfantis, E. A. and Au, M., *Efficient Image Compression Algorithm for Computer Animated Images*, Journal of Electronic Imaging, pp. 381-387, Oct 1992.

[36] Young, R. K., *Wavelet Theory and its Application*, Kluwer Academic Publishers, 1992.

[37] Ziv, J., and Lempel, A., *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, Vol. IT-23(3), May 1977.